Shell 13 问

wizardforcel

Published with GitBook



目錄

shell十三问--shell教程（markdown 版本）

> 作者：wzb56
>
> 来源：13_questions_of_shell

shell十三问之1: 何为shell？

`shell`是什么东西之前，不妨让我们重新审视`使用者`和`计算机系统`的关系：（此处为使用者和计算机系统的关系图）

我们知道计算机的运作不能离开硬件，但使用者却无法直接操作硬件， 硬件的驱动只能通过一种称为“`操作系统`(`OS`，`Opertating System`)”的软件来管控。 事实上，我们每天所谈的“`linux`”，严格来说只是一个`操作系统`(`OS`), 我们称之为“`内核`(`kernel`)”。

然而，从使用者的角度来说，使用者没有办法直接操作一个`kernel`， 而是通过`kernel`的“外壳”程序，也就是所谓的`shell`，来与`kernel`沟通。 这也正是`kernel`跟`shell`的形象命名的的关系。如图： （此处为kernel-->shell关系图；)

从技术的角度来说，`shell`是一个使用者与系统的`交互界面(interface)`, 只能让使用者通过`命令行`(`command line`)来使用系统来完成工作。 因此，`shell`最简单的定义就是----`命令解释器`(`Command Interpreter`):

- 将使用者的命令翻译给kernel来处理；
- 同时，将kernel的处理结果翻译给使用者。

每次当我们完成`系统登入`(`login`), 我们就取得一个交互模式的shell， 也称之为`login shell`或者`primary shell`。

若从`进程`(`process`)的角度来说，我们在shell所下达的命令，均是shell所产生的`子进程`。 这种现象，我暂可称之为`fork`。

如果是执行`shell脚本`(`shell script`)的话，脚本中命令则是由另一个非交互模式的`子shell`(`sub shell`)来执行的。 也就是primary shell产生sub shell的进程，而该sub shell 进程再产生script中所有命令的进程。 (关于进程，我们日后有机会在补充)

这里， 我们必须知道：`kernel` 与 `shell` 是不同的两套软件，而且都是可以被替换的：

- 不同的`OS`使用不同的`kernel`;
- 同一个kernel之上，也可以使用不同的`shell`;

在`Linux`的预设系统中，通常可以找到好几种不同的`shell`, 且通常会被记录在如下文件中：

```
/etc/shells
```

不同的`shell`有着不同的功能，且彼此各异，或者说“大同小异”。 常见的`shell`主要分为两大主流：

1. sh：

- burne shell (sh)
- burne again shell (bash)

2. csh：
   - c shell (csh)
   - tc shell (tcsh)
   - korn shell (ksh) (FIXME)

大部分的Linux操作系统的预设shell都是`bash`，其原因大致如下两种：

- 自由软件
- 功能强大 bash是gnu project最成功的产品之一，自推出以来深受广大`Unix`用户的喜爱， 且也逐渐成为不少组织的系统标准。

shell十三问之2：shell prompt(PS1)与Carriage Return(CR)关系

当你成功登陆一个shell终端的文字界面之后，大部分的情形下， 你会在屏幕上看到一个不断闪烁的方块或者底线(视不同的版本而别)， 我们称之为`游标`(`cursor`). `cursor`作用就是告诉你接下来你从键盘输入的按键所插入的位置， 且每输入一个键， `cursor`便向右移动一个格子， 如果连续输入太多的话，则自动接在下一行输入。

假如你刚完成登陆，还没有输入任何按键之前， 你所看到的`cursor`所在的位置的同一行的左边部分，我们称之为`提示符`(`prompt`)。

`提示符`的格式或因不同的版本而各有不同， 在Linux上，只需留意最接近`游标`的一个提示符号，通常是如下两者之一：

- $: 给一般用户账号使用;
- #: 给root(管理员)账号使用;

事实上，shell prompt的意思很简单： 告诉shell使用者，您现在可以输入命令行了。

我们可以说，使用者只有在得到shell prompt才能打命令行， 而`cursor`是指示键盘在命令行的输入位置，使用者每输入一个键， `cursor`就往后移动一个格，直到碰到命令行读进`CR`(`Carriage Return`, 由`Enter`键产生)字符为止。 `CR`的意思也很简单： 使用者告诉shell：老兄，你可以执行的我命令行了。 严格来说： 所谓的命令行， 就是在`shell prompt`与`CR`之间所输入的文字。

(**question**：为何我们这里坚持使用`CR`字符而不说`Enter`按键呢？ 答案在后面的学习中给出)。

不同的命令可以接受的命令的格式各有不同， 一般情况下，一个标准的命令行格式为如下所列：

```
command-name options argument
```

若从技术的细节上来看， shell会依据`IFS`(`Internal Field Seperator`)将command line 所输入的文字给拆解为`字段`(`word`). 然后在针对特殊的字符(meta)先做处理，最后在重组整行command line。

(注意：请务必理解以上两句的意思，我们日后的学习中常回到这里思考。)

其中`IFS`是shell预设使用的字段位分隔符号，可以由一个及多个如下按键组成：

- 空白键(White Space)

- 表格键(Tab)
- 回车键(Enter)

系统可以接受的命令的名称(command-name)可以从如下途径获得：

- 确定的路径所指定的外部命令
- 命令的别名(alias)
- shell内建命令(built-in)
- $PATH之下的外部命令

每一个命令行均必须包含命令的名称，这是不能缺少的。

shell十三问之3：别人echo、你也echo，是问echo知多少？

承接上一章介绍的 `command line` , 这里我们用 `echo` 这个命令加以进一步说明。

> 温习 标准的 `command line` 三个组成部分： `command_name option argument`

`echo` 是一个非常简单、直接的Linux命令：

```
$echo argument
```

`echo` 将argument送出到 `标准输出` (`stdout`),通常是在监视器(monitor)上输出。

> **Note**：
>
> 在linux系统中任何一个进程默认打开三个文件：stdin、stdout、stderr.
>
> stdin 标准输入
>
> stdout 标准输出
>
> stderr 标准错误输出

为了更好理解，不如先让我们先跑一下 `echo` 命令好了：

```
$echo

$
```

你会发现只有一个空白行，然后又回到了 `shell prompt` 上了。 这是因为 `echo` 在预设上，在显示完argument之后，还会送出以一个换行符号 (`new-line charactor`). 但是上面的command `echo` 并没有任何argument，那结果就只剩一个换行符号。 若你要取消这个换行符号， 可以利用 `echo` 的 `-n` 选项:

```
$echo -n
$
```

不妨让我们回到 `command line` 的概念上来讨论上例的echo命令好了： `command line` 只有command_name(`echo`)及option(`-n`),并没有显示任何 `argument` 。

要想看看 `echo` 的 `argument` ，那还不简单接下来，你可以试试如下的输入：

```
$echo first line
first line
$echo -n first line
first line $
```

以上两个 `echo` 命令中，你会发现 `argument` 的部分显示在你的屏幕， 而换行符则视 `-n` 选项的有无而别。 很明显的，第二个 `echo` 由于换行符被取消了， 接下来的 `shell prompt` 就接在输出结果的同一行了... ^_^。

事实上， `echo` 除了 `-n` 选项之外，常用选项有：

- -e: 启用反斜杠控制字符的转换(参考下表)
- -E: 关闭反斜杠控制字符的转换(预设如此)
- -n: 取消行末的换行符号(与-e选项下的\c字符同意)

关于 `echo` 命令所支持的反斜杠控制字符如下表：

	转义字符	字符的意义
	\a	ALERT / BELL(从系统的喇叭送出铃声)
	\b	BACKSPACE, 也就是向左退格键
	\c	取消行末之换行符号
	\E	ESCAPE, 脱字符键
	\f	FORMFEED, 换页字符
	\n	NEWLINE, 换行字符
	\r	RETURN, 回车键
	\t	TAB, 表格跳位键
	\v	VERTICAL TAB, 垂直表格跳位键
	\n	ASCII 八进制编码(以x开头的为十六进制)，此处的n为数字
	\	反斜杠本身

> **Note**： 上述表格的资料来自O'Reilly出版社的**Learning the Bash Shell, 2nd Ed**.

或许，我们可以通过实例来了解 `echo` 的选项及控制字符：

例一：

```
$ echo -e "a\tb\tc\n\d\te\tf"
a    b    c
d    e    f
$
```

上例中，用\t来分割abc还有def，及用\n将def换至下一行。

例二：

```
$echo -e "\141\011\142\011\143\012\144\011\145\011\146"
a    b    c
d    e    f
```

与例一中结果一样，只是使用ASCII八进制编码。

例三：

```
$echo -e "\x61\x09\x62\x09\x63\x0a\x64\x09\x65\x09\x66"
a    b    c
d    e    f
```

与例二差不多，只是这次换用ASCII的十六进制编码。

例四：

```
$echo -ne "a\tb\tc\nd\te\bf\a"
a       b       c
d       f $
```

因为e字母后面是退格键(\b)，因此输出结果就没有e了。 在结束的时听到一声铃响，是\a的杰作。 由于同时使用了-n选项，因此 `shell prompt` 紧接在第二行之后。 若你不用-n的话，那你在\a后再加个\c，也是同样的效果。

事实上，在日后的 `shell` 操作及 `shell script` 设计上， `echo` 命令是最常被使用的命令之一。 比方说，使用 `echo` 来检查变量值：

```
$ A=B
$ echo $A
B
$ echo $?
0
```

> **Note:** 关于变量的概念，我们留到以下的两章跟大家说明。

好了，更多的关于 `command line` 的格式， 以及 `echo` 命令的选项， 请您自行多加练习、运用了...

shell十三问之4："""(双引号)与''(单引号)差在哪？

还是回到我们的`command line`来吧...

经过前面两章的学习，应该很清楚当你在`shell prompt`后面敲打键盘, 直到按下`Enter`键的时候，你输入的文字就是`command line`了， 然后`shell`才会以进程的方式执行你所交给它的命令。 但是，你又可知道：你在`command line`中输入的每一个文字， 对`shell`来说，是有类别之分的呢？

简单而言，(我不敢说精确的定义，注1), `command line`的每一个`charactor`, 分为如下两种：

- literal：也就是普通的纯文字，对`shell`来说没特殊功能；
- meta: 对`shell`来说，具有特定功能的特殊保留元字符。

> **Note:**
>
> 对于`bash shell`在处理`comamnd line`的顺序说明， 请参考O'Reilly出版社的**Learning the Bash Shell**，**2nd Edition**， 第177-180页的说明，尤其是178页的流程图：Figure 7-1 ...

`literal`没什么好谈的， 像abcd、123456这些"文字"都是literal...(so easy? ^_^) 但meta却常使我们困惑...(confused?) 事实上，前两章，我们在`command line`中已碰到两个 似乎每次都会碰到的meta：

- `IFS`：有`space`或者`tab`或者`Enter`三者之一组成(我们常用space)
- `CR`：由`Enter`产生；

`IFS`是用来拆解`command line`中每一个词(word)用的， 因为`shell command line`是按词来处理的。 而`CR`则是用来结束`command line`用的，这也是为何我们敲`Enter`键， 命令就会跑的原因。

除了常用的`IFS`与`CR`, 常用的meta还有：

meta字符	meta字符作用		
=	设定变量		
$	作变量或运算替换(请不要与 `shell prompt` 混淆)	命令	
>	输出重定向(重定向stdout)		
<	输入重定向(重定向stdin)		
\		命令管道	
&	重定向file descriptor或将命令至于后台(bg)运行		
()	将其内部的命令置于nested subshell执行，或用于运算或变量替换		
{}	将期内的命令置于non-named function中执行，或用在变量替换的界定范围		
;	在前一个命令执行结束时，而忽略其返回值，继续执行下一个命令		
&&	在前一个命令执行结束时，若返回值为true，继续执行下一个命令		
\	\		在前一个命令执行结束时，若返回值为false，继续执行下一个命令
!	执行histroy列表中的命令		
...	...		

假如我们需要在 `command line` 中将这些保留元字符的功能关闭的话， 就需要quoting处理了。

在 `bash` 中，常用的quoting有以下三种方法：

- hard quote：''(单引号)，凡在hard quote中的所有meta均被关闭；
- soft quote：""(双引号)，凡在soft quote中大部分meta都会被关闭，但某些会保留(如$);
- escape: \ (反斜杠)，只有在紧接在escape(跳脱字符)之后的单一meta才被关闭；

> **Note:**
>
> 在soft quote中被豁免的具体meta清单，我不完全知道， 有待大家补充，或通过实践来发现并理解。

下面的例子将有助于我们对quoting的了解：

```
$ A=B C #空白符未被关闭，作为IFS处理
$ C：command not found.
$ echo $A

$ A="B C" #空白符已被关掉，仅作为空白符
$ echo $A
B C
```

在第一个给A变量赋值时，由于空白符没有被关闭， `command line` 将被解释为： `A=B 然后碰到<IFS>，接着执行C命令` 在第二次给A变量赋值时，由于空白符被置于soft quote中，因此被关闭，不在作为 `IFS` ； `A=B<space>C` 事实上，空白符无论在soft quote还是在hard quote中， 均被关闭。Enter键字符亦然：

```
$ A=`B
> C
> '
$ echo "$A"
B
C
```

在上例中，由于 `enter` 被置于hard quote当中，因此不再作为 `CR` 字符来处理。 这里的 `enter` 单纯只是一个断行符号(new-line)而已， 由于 `command line` 并没得到 `CR` 字符， 因此进入第二个 `shell prompt` (`PS2` , 以>符号表示)， `command line` 并不会结束，直到第三行，我们输入的 `enter` 并不在hard quote里面， 因此没有被关闭， 此时， `command line` 碰到 `CR` 字符，于是结束，交给shell来处理。

上例的 `Enter` 要是被置于soft quote中的话， `CR` 字符也会同样被关闭：

```
$ A="B
> C
> "
$ echo $A
B C
```

然而，由于 `echo $A` 时的变量没有置于soft quote中， 因此，当变量替换完成后，并作命令行重组时， `enter` 被解释为 `IFS` ， 而不是new-line字符。

同样的，用escape亦可关闭CR字符：

```
$ A=B\
> C\
>
$ echo $A
BC
```

上例中的，第一个 `enter` 跟第二个 `enter` 均被escape字符关闭了， 因此也不作为 `CR` 来处理，但第三个 `enter` 由于没有被escape， 因此，作为 `CR` 结束 `command line` 。 但由于 `enter` 键本身在shell meta中特殊性，在 \ escape字符后面 仅仅取消其 `CR` 功能， 而不保留其IFS功能。

你或许发现光是一个`enter`键所产生的字符，就有可能是如下这些可能：

- CR
- IFS
- NL(New Line)
- FF(Form Feed)
- NULL
- ...

至于，什么时候解释为什么字符，这个我就没法去挖掘了， 或者留给读者君自行慢慢摸索了...^-^

至于soft quote跟hard quote的不同，主要是对于某些meta的关闭与否，以$来做说明：

```
$ A=B\ C
$ echo "$A"
B C
$ echo '$A'
$A
```

在第一个`echo`命令行中，$被置于soft quote中，将不被关闭， 因此继续处理变量替换， 因此，`echo`将A的变量值输出到屏幕，也就是"B C"的结果。

在第二个`echo`命令行中，$被置于hard quote中，则被关闭， 因此，$只是一个$符号，并不会用来做变量替换处理， 因此结果是$符号后面接一个A字母：$A.

> 练习与思考: 如下结果为何不同？
>
> tips: 单引号和双引号，在quoting中均被关闭了。

```
$ A=B\ C
$ echo '"$A"'  #最外面的是单引号
"$A"
$ echo "'$A'"  #最外面的是双引号
'B C'
```

在CU的shell版里，我发现很多初学者的问题， 都与quoting的理解有关。 比方说，若我们在awk或sed的命令参数中， 调用之前设定的一些变量时，常会问及为何不能的问题。

要解决这些问题，关键点就是：区分出 **shell meta** 与 **command meta**

前面我们提到的那些meta，都是在command line中有特殊用途的， 比方说{}就是将一系列的command line置于不具名的函数中执行(可简单视为command block)， 但是，awk却需要用{}来区分出awk的命令区段(BEGIN,MAIN,END). 若你在command line中如此输入：

```
$ awk {print $0} 1.txt
```

由于{}在shell中并没有关闭，那shell就将{print $0}视为command block， 但同时没有 `;` 符号作命令分隔，因此，就出现awk语法错误结果。

要解决之，可用hard quote:

```
awk '{print $0}'
```

上面的hard quote应好理解，就是将原来的 {、 、$、}这几个shell meta关闭， 避免掉在shell中遭到处理，而完整的成为awk的参数中command meta。

> **Note:**
>
> awk中使用的$0 是awk中内建的field nubmer，而非awk的变量， awk自身的变量无需使用$.

要是理解了hard quote的功能，在来理解soft quote与escape就不难：

```
awk "{print \$0}" 1.txt
awk \{print \$0\} 1.txt
```

然而，若要你改变awk的$0的0值是从另一个shell变量中读进呢？ 比方说：已有变量$A的值是0， 那如何在 `command line` 中解决 awk的$$A呢？ 你可以很直接否定掉hard quote的方案：

```
$ awk '{print $$A}' 1.txt
```

那是因为$A的$在hard quote中是不能替换变量的。

聪明的读者(如你！)，经过本章的学习，我想，你应该可以理解为 为何我们可以使用如下操作了吧：

```
A=0
awk "{print \$$A}" 1.txt
awk  \{print\ \$$A\} 1.txt
awk '{print $'$A'}' 1.txt
awk '{print $'"$A"'}' 1.txt
```

或许，你能给出更多方案... ^_^

更多练习：

- http://bbs.chinaunix.net/forum/viewtopic.php?t=207178 一个关于read命令的小问题： 很早以前觉得很奇怪：执行read命令，然后读取用户输入给变量赋值， 但如果输入是以空格键开始的话，这空格会被忽略，比如：

  ```
  read a  #输入：    abc
  echo "$a" #只输出abc
  ```

原因: 变量a的值，从终端输入的值是以IFS开头，而这些IFS将被shell解释器忽略(trim)。应该与shell解释器分词的规则有关；

```
read a  #输入：\ \ \ abc
echo "$a" #只输出abc
```

需要将空格字符转义

> **Note:**
>
> IFS Internal field separators, normally space, tab, and newline (see Blank Interpretation section). ...... Blank Interpretation After parameter and command substitution, the results of substitution
> are scanned for internal field separator characters (those found in IFS) and split into distinct arguments where such characters are found. Explicit null arguments ("" or '') are retained.
> Implicit null arguments(those resulting from parameters that have no values) are removed. (refre to: man sh)

解决思路：

1. shell command line 主要是将整行line给分解(break down)为每一个单词(word);
2. 而词与词之间的分隔符就是IFS (Internal Field Seperator)。
3. shell会对command line作处理(如替换，quoting等), 然后再按词重组。(注：别忘了这个重组特性)
4. 当你用IFS来事开头一个变量值，那shell会先整理出这个词，然后在重组command line。5.然而，你将IFS换成其他，那shell将视你哪些space/tab为“词”，而不是IFS。那在重组时，可以得到这些词。

若你还是不理解，那来验证一下下面这个例子：

```
$ A="  abc"
$ echo $A
abc
$ echo "$A" #note1
   abc
$ old_IFS=$IFS
$ IFS=;
$ echo $A
   abc
$ IFS=$old_IFS
$ echo $A
abc
```

> **Note:**
>
> 1. 这里是用 soft quoting 将里面的 space 关闭，使之不是 meta(IFS)， 而是一个 literal(white space);
> 2. IFS=; 意义是将IFS设置为空字符，因为;是shell的元字符(meta);

问题二：为什么多做了几个分号，我想知道为什么会出现空格呢？

```
$ a=";;;test"
$ IFS=";"
$ echo $a
   test
$ a="   test"
$ echo $a
   test
$ IFS=" "
$ echo $a
test
```

解答：

这个问题，出在 `IFS=;` 上。 因为这个 `;` 在问题一中的command line上是一个meta, 并非 `";"` 符号本身。 因此， `IFS=;` 是将IFS设置为 null charactor (不是space、tab、newline)。

要不是试试下面这个代码片段：

```
$ old_IFS=$IFS
$ read A
;a;b;c
$ echo $A
;a;b;c
$ IFS=";"  #Note2
$ echo $A
a b c
```

> **Note:**
>
> 要关闭 `;` 可用 `";"` 或者 `';'` 或者 `\;` 。

- http://bbs.chinaunix.net/forum/viewtopic.php?t=216729

思考问题二：文本处理：读文件时，如何保证原汁原味。

```
cat file | while read i
do
   echo $i
done
```

文件file的行中包含若干空，经过read只保留不重复的空格。 如何才能所见即所得。

```
cat file | while read i
do
   echo "X${i}X"
done
```

从上面的输出，可以看出read，读入是按整行读入的; 不能原汁原味的原因：

1. 如果行的起始部分有IFS之类的字符，将被忽略;
2. `echo $i` 的解析过程中，首先将$i替换为字符串， 然后对echo 字符串中字符串分词，然后命令重组，输出结果; 在分词，与命令重组时，可能导致多个相邻的IFS转化为一个;

```
cat file | while read i
do
  echo "$i"
done
```

以上代码可以解决原因2中的，command line的分词和重组导致meta字符丢失； 但仍然解决不了原因1中，read读取行时，忽略行起始的IFS meta字符。

回过头来看上面这个问题：为何要原汁原味呢？ cat命令就是原汁原味的，只是shell的read、echo导致了某些shell的meta字符丢失;

如果只是IFS meta的丢失，可以采用如下方式： 将IFS设置为null，即 `IFS=;` , 在此再次重申此处 `;` 是shell的meta字符,而不是literal字符; 因此要使用literal的 `;` 应该是 `\;` 或者关闭meta 的(soft/hard) quoting的 `";"` 或者 `';'` 。

因此上述的解决方案是：

```
old_IFS=$IFS
IFS=; #将IFS设置为null
cat file | while read i
do
  echo "$i"
done
IFS=old_IFS #恢复IFS的原始值
```

现在，回过头来看这个问题，为什么会有这个问题呢； 其本源的问题应该是没有找到解决原始问题的最合适的方法， 而是采取了一个迂回的方式来解决了问题；

因此，我们应该回到问题的本源，重新审视一下，问题的本质。 如果要精准的获取文件的内容，应该使用od或者hexdump会更好些。

shell十三问之5:问var=value 在export前后的差在哪?

这次让我们暂时丢开`command line`, 先了解一下bash变量(variable)吧...

所谓的变量，就是利用一个固定的"名称"(name), 来存取一段可以变化的"值"(value)。

1. 变量设定(set)

在bash中， 你可以用"="来设定或者重新定义变量的内容：

```
name=value
```

在设定变量的时候，得遵守如下规则：

- 等号左右两边不能使用分隔符号(IFS),也应避免使用shell的保留元字符(meta charactor);
- 变量的名称(name)不能使用$符号;
- 变量的名称(name)的首字符不能是数字(number)。
- 变量的名称(name)的长度不可超过256个字符。
- 变量的名称(name)及变量的值的大小写是有区别的、敏感的(case sensitive，)

如下是一些变量设定时常见的错误：

```
A= B #=号前后不能有IFS
1A=B #变量名称不能以数字开头
$A=B #变量的名称里有$
a=B  #这跟a=b是不同的,(这不是错误，提醒windows用户)
```

如下则是可以接受的设定：

```
A=" B" #IFS被关闭，参考前面的quoting章节
A1=B   #并非以数字开头
A=$B   #$可用在变量的值内
This_Is_A_Long_Name=b #可用_连接较长的名称或值，且有大小区别；
```

2. 变量替换(substitution)

shell 之所以强大，其中的一个因素是它可以在命令行中对变量作 替换(substitution)处理。 在命令行中使用者可以使用$符号加上变量名称(除了用=定义变量名称之外)， 将变量值给替换出来，然后再重新组建命令行。

比方:

```
$ A=ls
$ B=la
$ C=/tmp
$ $A -$B $C
```

以上命令行的第一个 `$` 是 `shell prompt` , 并不在命令行之内。 必须强调的是，我们所提的变量替换，只发生在 `command line` 上面。 (是的，请让我们再次回到命令行吧！) 仔细分析,最后那行 `command line` ,不难发现在被执行前(在输入 `CR` 字符之前)， `$` 符号对每一个变量作替换处理(将变量的值替换出来再重组命令行), 最后会得出如下命令行：

```
ls -la /tmp
```

还记得第二章，我请大家"务必理解"的那两句吗？ 若你忘了，我这里重贴一遍：

> **Note:**
>
> 若从技术的细节来看， `shell` 会依据 `IFS` (Internal Field Seperator) 将 `command line` 所输入的文字拆解为"字段"(word/field)。 然后再针对特殊字符(meta)先作处理，最后重组整行 `command line` 。

这里的 `$` 就是 `command line` 中最经典的meta之一了， 就是作变量替换的。在日常的shell操作中， 我们常会使用 `echo` 命令来查看特定的变量的值， 例如：

```
$ echo $A -$B $C
```

我们已学过， `echo` 命令只单纯将其argument送至"标准输出"(stdout, 通常是我们的屏幕)。 所以上面的命令会在屏幕上得到如下结果：

```
ls -al /tmp
```

这是由于 `echo` 命令在执行时，会先将 `$A` (ls)、 `$B` (la)跟 `$C` (/tmp)给替换出来； 利用shell对变量的替换处理能力，我们在设定变量时就更为灵活了：

```
A=B
B=$A
```

这样，B的变量值就可继承A变量"当时"的变量值了。 不过，不要以"数学逻辑"来套用变量的设定，比方说：

```
A=B
B=C
```

这样，并不会让A的变量值变成C。再如：

```
A=B
B=$A
A=C
```

同样也不会让B的值变成C。

上面是单纯定义了两个不同名称的变量：A 与 B, 它们的取值分别是C与B。

若变量被重复定义的话，则原有值为新值所取代。(这不正是"可变的量"吗？^_^) 当我们在设定变量的时候，请记住这点：用一个名称存储一个数值， 仅此而已。

此外， 我们也可以利用命令行的变量替换能力来"扩充"(append)变量的值：

```
A=B:C:D
A=$A:E
```

这样， 第一行我们设定A的值为"B:C:D", 然后,第二行再将值扩充为"B:C:D:E"。

上面的扩充的范例，我们使用分隔符号(:)来达到扩充的目的， 要是没有分隔符的话，如下是有问题的：

```
A=BCD
B=$AE
```

因为第二次是将A的值继承$AE的替换结果，而非$A再加E。 要解决此问题，我们可用更严谨的替换处理：

```
A=BCD
A=${A}E
```

上例中，我们使用{}将变量名称范围给明确定义出来， 如此一来， 我们就可以将A的变量值从BCD给扩充为BCDE。

> **Tips:** 关于${name}事实上还可以做到更多的变量处理能力， 这些均属于比较进阶阶段的变量处理，现阶段暂不介绍了， 请大家自行参考资料。

3. export 变量

严格来说，我们在当前shell中所定义的变量，均属于 "本地变量"(local variable), 只有经过 `export` 命令的 "输出"处理，才能成为"环境变量"(environment variable)：

```
$ A=B
$ export A
```

或者

```
$ export A=B
```

经过 `export` 输出处理之后，变量A就能成为一个环境变量 供其后的命令使用。在使用 `export` 的时候，请别忘记 shell在命令行对变量的"替换"(substitution)处理。 比方说：

```
$ A=B
$ B=C
$ export $A
```

上面的命令并未将A输出为"环境变量"，而是将B导出 这是因为在这个命令行中，$A会首先被替换为B,然后在"塞回" 作 `export` 的参数。

要理解这个 `export` ，事实上需要从process(进程)的角度来理解才能透彻。 我们将于下一章为大家说明process(进程)的概念，敬请留意。

4. 取消变量(unset)

要取消一个变量，在bash中可使用 `unset` 命令来处理：

```
    unset A
```

与 `export` 一样， `unset` 命令行，也同样会作 变量替换(这其实是shell的功能之一)， 因此:

```
$ A=B
$ B=C
$ unset $A
```

事实上，所取消的是变量B而不是A。

此外，变量一旦经过unset取消之后， 其结果是将整个变量拿掉，而不是取消变量的值。

如下两行其实是很不一样的：

```
$ A=
$ unset A
```

第一行只是将变量A设定为"空值"(null value), 但第二行则是让变量A不存在。 虽然用眼睛来看， 这两种变量的状态在如下的命令结果中都是一样的：

```
$ A=
$ echo $A

$ unset A
$ echo $A
```

请学员务必能识别null value 与 unset的本质区别， 这在一些进阶的变量处理上是很严格的。

比方说：

```
$ str=    #设为null
$ var=${str=expr} #定义var
$ echo $var

$ echo $str

$ unset str #取消str
$ var=${str=expr} #定义var
$ echo $var
expr
$ echo $str
expr
```

聪明的读者(yes, you!)，稍加思考的话， 应该不难发现为何同样的var=${str=expr} 在str为null与unset之下的不同吧？ 若你看不出来，那可能是如下原因之一：

- 你太笨了
- 不了解 var=${str=expr} 这个进阶处理
- 对本篇说明还没有来得及消化吸收
- 我讲得不好

不知，您选哪个呢？...... ^_^.

shell十三问之6：exec跟source差在哪？

这次让我们从CU shell版的一个实例帖子来谈起吧： (论坛改版后，原链接已经失效)

例中的提问原文如下：

> 帖子提问:
>
> cd /etc/aa/bb/cc可以执行 但是把这条命令放入shell脚本后，shell脚本不执行！ 这是什么原因？

意思是：运行shell脚本，并没有移动到/etc/aa/bb/cc目录。

我当时如何回答暂时别去深究，先让我们了解一下进程 (process)的概念好了。

首先，我们所执行的任何程序，都是父进程(parent process)产生的一个 子进程(child process),子进程在结束后，将返回到父进程去。 此现象在Linux中被称为 `fork` 。

(为何要称为fork呢？ 嗯，画一下图或许比较好理解...^_^)

当子进程被产生的时候，将会从父进程那里获得一定的资源分配、及 (更重要的是)继承父进程的环境。

让我们回到上一章所谈到的"环境变量"吧： 所谓环境变量其实就是那些会传给子进程的变量。 简单而言, "遗传性"就是区分本地变量与环境变量的决定性指标。 然而，从遗传的角度来看，我们不难发现环境变量的另一个重要特征： 环境变量只能从父进程到子进程单向传递。 换句话说：在子进程中环境如何变更，均不会影响父进程的环境。

接下来，在让我们了解一下shell脚本(shell script)的概念. 所谓shell script 讲起来很简单，就是将你平时在shell prompt输入的多行 `command line` , 依序输入到一个文件文件而已。

再结合以上两个概念(process + script)，那应该不难理解如下的这句话的意思了： 正常来说，当我们执行一个shell script时，其实是先产生一个sub-shell的子进程， 然后sub-shell再去产生命令行的子进程。 然则，那让我们回到本章开始时，所提到的例子在重新思考：

> 帖子提问:
>
> cd /etc/aa/bb/cc可以执行 但是把这条命令放入shell脚本后，shell脚本不执行！ 这是什么原因？

意思是：运行shell脚本，并没有移动到/etc/aa/bb/cc目录。

我当时的答案是这样的：

> 因为，我们一般跑的shell script是用sub-shell去执行的。 从process的概念来看，是parent process产生一个child process去执行， 当child结束后，返回parent, 但parent的环境是不会因child的改变而改变的。 所谓的环境变量元数很多，如effective id(euid)，variable, working dir等等... 其中的working dir($PWD) 正是楼主的疑问所在： 当用sub-shell来跑script的话，sub-shell的$pwd会因为cd而变更， 但返回primary shell时，$PWD是不会变更的。

能够了解问题的原因及其原理是很好的，但是？ 如何解决问题，恐怕是我们更应该感兴趣的是吧？

那好，接下来，再让我们了解一下 `source` 命令好了。 当你有了 `fork` 的概念之后，要理解 `soruce` 就不难：

所谓 `source` ，就是让script在当前shell内执行、 而不是产生一个sub-shell来执行。 由于所有执行结果均在当前shell内执行、而不是产生一个sub-shell来执行。

因此, 只要我们原本单独输入的script命令行，变成 `source` 命令的参数， 就可轻而易举地解决前面提到的问题了。

比方说，原本我们是如此执行script的：

```
$ ./my_script.sh
```

现在改成这样既可：

```
$ source ./my_script.sh
```

或者：

```
$ . ./my_script.sh
```

说到这里，我想，各位有兴趣看看 `/etc` 底下的众多设定的文件， 应该不难理解它们被定义后，如何让其他script读取并继承了吧？

若然，日后，你有机会写自己的script， 应也不难专门指定一个设定的文件以供不同的script一起"共用"了... ^_^

okay,到这里，若你搞懂 `fork` 与 `source` 的不同， 那接下来再接受一个挑战：

> 那 `exec` 又与 `source` / `fork` 有何不同呢？

哦...要了解 `exec` 或许较为复杂，尤其是扯上 `File Decscriptor` 的话... 不过，简单来说：

> `exec` 也是让script在同一个进程上执行，但是原有进程则被结束了。 简言之，原有进程能否终止，就是 `exec` 与 `source` / `fork` 的最大差异了。

嗯，光是从理论去理解，或许没那么好消化， 不如动手"实践+思考"来得印象深刻哦。

下面让我们为两个简单的script，分别命名为1.sh以及2.sh

1.sh

```
#!/bin/bash

A=B
echo "PID for 1.sh before exec/source/fork:$$"

export A
echo "1.sh: \$A is $A"

case $1 in
        exec)
                echo "using exec..."
                exec ./2.sh ;;
        source)
                echo "using source..."
                . ./2.sh ;;
        *)
                echo "using fork by default..."
                ./2.sh ;;
esac

echo "PID for 1.sh after exec/source/fork:$$"
echo "1.sh: \$A is $A"
```

2.sh

```
#!/bin/bash

echo "PID for 2.sh: $$"
echo "2.sh get \$A=$A from 1.sh"

A=C
export A
echo "2.sh: \$A is $A"
```

然后分别跑如下参数来观察结果：

```
$ ./1.sh fork
$ ./1.sh source
$ ./1.sh exec
```

好了，别忘了仔细比较输出结果的不同及背后的原因哦... 若有疑问，欢迎提出来一起讨论讨论~~~~

happy scripting！ ^_^

shell十三问之7：()与{}差在哪？

嗯，这次轻松一下，不讲太多... ^_^

先说一下，为何要用()或者{}好了。

许多时候，我们在shell操作上，需要在 一定的条件下执行多个命令，也就是说， 要么不执行，要么就全执行，而不是每次 依序的判断是否要执行下一个命令。

或者，要从一些命令执行的先后次序中得到结果， 如算术运算的2*(3+4)那样...

这时候，我们就可以引入"`命令群组`"(`command group`)的概念：将许多命令集中处理。

在shell `command line` 中，一般人或许不太计较 `()` 与 `{}` 这两对符号的差异，虽然两者都可以将多个命令当作群组处理， 但若从技术细节上，却是很不一样的：

- `()` 将 `command group` 置于 `sub-shell` (`子shell`)中去执行，也称 `nested sub-shell`。
- `{}` 则是在同一个 `shell` 内完成，也称 `non-named command group`。

若你对上一章的fork与source的概念还记得的话， 那就不难理解两者的差异了。

要是在 `command group` 中扯上变量及其他环境的修改， 我们可以根据不同的需求来使用 `()` 或 `{}`。 通常而言, 若所作的修改是临时的，且不想影响原有或以后的设定， 那我们就使用 `nested sub-shell`，即 `()`；反之，则用 `non-named command group`，即 `{}`。

是的，光从 `command line` 来看，`()` 与 `{}` 差别就讲完了，够轻松吧~~~, ^_^

然而，这两个 `meta` 用在其他 `command meta` 或领域中(如Regular Expression)， 还是有很多差别的。 只是，我不打算再去说明了，留给读者慢慢发掘好了...

我这里只想补充一个概念，就是 `function`。 所谓 `function`，就是用一个名字去命名一个 `command group`，然后再调用这个名字去执行 `command group`。

从 `non-named command group` 来推断， 大概你也可以推测到我要说的是 `{}` 了吧？(yes! 你眞聪明 ^_^)

在bash中，function的定义方式有两种：

- 方式一：

```
function function_name {
  command1
  command2
  command3
  .....
}
```

- 方式二：

```
function_name () {
  command1
  command2
  command3
  ......
}
```

用哪一种方式无所谓， 只是碰到所定义的名称与现有的命令或者别名冲突的话， 方式二或许会失败。 但方式二起码可以少打个 `function` 这一串英文字符， 对懒人来说(如我)，有何乐而不为呢？...^_^

`function` 在一定程度上来说，也可以称为"函数"， 但请不要与传统编程所使用的"函数"(library)搞混了， 毕竟两者差异很大。 唯一相同的是，我们都可以随时用"已定义的名称"来调用它们...

若我们在shell操作中，需要不断地重复某些命令， 我们首先想到的，或许是将命令写成shell脚本(shell script)。 不过，我们也可以写成function, 然后在command line中打上function_name就可当一般的shell script使用了。

若只是你在shell中定义的 `function` , 除了用 `unset` function_name取消外， 一旦你退出shell， function也跟着消失。 然而，在script中使用function却有许多好处， 除了提高整体script的执行性能外(因为已经载入)， 还可以节省许多重复的代码......

简单而言，若你会将多个命令写成script以供调用的话， 那你可以将function看成script中script。... ^_^

而且通过上一章节介绍的 `source` 命令， 我们可以自行定义许许多多好用的function， 在集中写在特定文件中， 然后，在其他的script中用 `source` 将它们载入，并反复执行。

若你是 `RedHat Linux` 的使用者， 或许，已经猜出 `/etc/rc.d/init.d/functions` 这个文件时啥作用了~~~ ^_^

okay，说要轻松点的嘛，那这次就暂时写到这吧。 祝大家学习愉快，^_^

shell十三问之8: $(())与$()还有${}差在哪？

我们上一章介绍了()与{}的不同， 这次让我们扩展一下，看看更多的变化：$()与${}又是啥玩意儿呢？

在bash shell中，`$()`与``(反引号)都是用来做`命令替换`(command substitution)的。

所谓的`命令替换`与我们第五章学过的变量替换差不多， 都是用来`重组命令行`： 完成`` 或者`$()`里面的 命令，将其结果替换出来， 再重组命令行。

例如：

```
$ echo the last sunday is $(date -d "last sunday" +%Y-%m-%d)
```

如此便可方便得到上一个星期天的日期了...^_^

在操作上， 用$()或``都无所谓, 只是我个人比较喜欢用$(),理由是：

1. ``(反引号)很容易与''(单引号)搞混乱，尤其对初学者来说。 有时在一些奇怪的字形显示中，两种符号是一模一样的(只取两点)。 当然了有经验的朋友还是一眼就能分辨两者。只是，若能更好的避免混乱， 又何乐而不为呢？ ^_^

2. 在多次的复合替换中， ``需要额外的转义(escape,)处理，而$()则比较直观。 例如，一个错误的使用的例子：

   ```
   command1 `command2 `command3` `
   ```

   原来的本意是要在command2 `command3` , 先将command3替换出来给command2处理， 然后再将command2的处理结果，给command1来处理。 然而真正的结果在命令行中却是分成了`command2`与 ``。

正确的输入应该如下：

```
  command1 `command2 \`command3\` `
```

要不然换成$()就没有问题了：

```
    command1 $(commmand2 $(command3))
```

只要你喜欢，做多少层的替换都没有问题~~~^_^

不过，$()并不是没有弊端的... 首先，``基本上可用在所有的unix shell中使用， 若写成 shell script，其移植性比较高。 而$()并不是每一种shell都能使用，我只能说， 若你用bash2的话，肯定没问题... ^_^

接下来，再让我们看看${}吧...它其实就是用来做 变量替换用的啦。 一般情况下，$var与${var}并没有啥不一样。 但是用${}会比较精准的界定变量名称的范围， 比方说:

```
$ A=B
$ echo $AB
```

原本是打算先将$A的结果替换出来， 然后在其后补一个字母B；但命令行上， 眞正的结果却是替换变量名称为AB的值出来... 若使用${}就没有问题了：

```
$ A=B
$ echo ${A}B
$ BB
```

不过，假如你只看到 `${}` 只能用来界定变量名称的话， 那你就实在太小看bash了。

为了完整起见，我这里再用一些例子加以说明 `${}` 的一些 特异功能： 假设我们定义了一个变量file为：

```
file=/dir1/dir2/dir3/my.file.txt
```

我们可以用 `${}` 分别替换获得不同的值：

1. shell字符串的非贪婪(最小匹配)左删除

```
${file#*/}  #其值为：dir1/dir2/dir3/my.file.txt
```

拿掉第一个 `/` 及其左边的字符串，其结果为： `dir1/dir2/dir3/my.file.txt` 。

```
${file#*.}  #其值为：file.txt
```

拿掉第一个 `.` 及其左边的字符串，其结果为： `file.txt` 。

2. shell字符串的贪婪(最大匹配)左删除：

```
${file##*/} #其值为：my.file.txt
```

拿掉最后一个 `/` 及其左边的字符串，其结果为： `my.file.txt`

```
${file##*.} #其值为：txt
```

拿掉最后一个 `.` 及其左边的字符串，其结果为： `txt`

3. shell字符串的非贪婪(最小匹配)右删除：

```
${file%/*}  #其值为：/dir1/dir2/dir3
```

拿掉最后一个 `/` 及其右边的字符串，其结果为： `/dir1/dir2/dir3` 。

```
${file%.*}  #其值为：/dir1/dir2/dir3/my.file
```

拿掉最后一个 `.` 及其右边的字符串，其结果为： `/dir1/dir2/dir3/my.file` 。

4. shell字符串的贪婪(最大匹配)右删除：

```
${file%%/*}  #其值为：其值为空。
```

拿掉第一个 `/` 及其右边的字符串，其结果为： 空串。

```
${file%%.*}  #其值为：/dir1/dir2/dir3/my。
```

拿掉第一个 `.` 及其右边的字符串，其结果为：/dir1/dir2/dir3/my。

> **Tips:**
>
> 记忆方法：
>
> `#` 是去掉左边(在键盘上 `#` 在 `$` 的左边);
>
> `%` 是去掉右边(在键盘上 `%` 在 `$` 的右边);
>
> 单个符号是最小匹配;
>
> 两个符号是最大匹配;

5. shell字符串取子串：

```
${file:0:5} #提取最左边的5个字符：/dir1
${file:5:5} #提取第5个字符及其右边的5个字符:/dir2
```

shell字符串取子串的格式：`${s:pos:length}`，取字符串s的子串：从pos位置开始的字符(包括该字符)的长度为length的的子串; 其中pos为子串的首字符，在s中位置；length为子串的长度;

> **Note:** 字符串中字符的起始编号为0.

6. shell字符串变量值的替换：

```
${file/dir/path}  #将第一个dir替换为path：/path1/dir2/dir3/my.file.txt
${file//dir/path} #将全部的dir替换为path：/path1/path2/path3/my.file.txt
```

shell字符串变量值的替换格式：

- 首次替换：`${s/src_pattern/dst_pattern}` 将字符串s中的第一个src_pattern替换为dst_pattern。
- 全部替换：`${s//src_pattern/dst_pattern}` 将字符串s中的所有出现的src_pattern替换为dst_pattern.

7. ${}还可针对变量的不同状态(没设定、空值、非空值)进行赋值：

- `${file-my.file.txt}` #如果file没有设定，则使用 使用my.file.txt作为返回值, 否则返回${file};(空值及非空值时，不作处理。);
- `${file:-my.file.txt}` #如果file没有设定或者${file}为空值, 均使用my.file.txt作为其返回值，否则，返回${file}.(${file} 为非空值时，不作处理);
- `${file+my.file.txt}` #如果file已设定(为空值或非空值), 则使用my.file.txt作为其返回值，否则不作处理。(未设定时，不作处理);
- `${file:+my.file.txt}` #如果${file}为非空值, 则使用my.file.txt作为其返回值，否则，(未设定或者为空值时)不作处理。
- `${file=my.file.txt}` #如果file为设定，则将file赋值为my.file.txt，同时将${file}作为其返回值；否则，file已设定(为空值或非空值)，则返回${file}。
- `${file:=my.file.txt}` #如果file未设定或者${file}为空值, 则my.file.txt作为其返回值， 同时，将${file}赋值为my.file.txt，否则，(非空值时)不作处理。

- `${file?my.file.txt}` #如果file没有设定，则将my.file.txt输出至STDERR, 否侧， 已设定(空值与非空值时)，不作处理。
- `${file:?my.file.txt}` #若果file未设定或者为空值，则将my.file.txt输出至STDERR，否则， 非空值时，不作任何处理。

> **Tips:**
>
> 以上的理解在于，你一定要分清楚，`unset` 与 `null` 以及non-null这三种状态的赋值；一般而言，与null有关，若不带 `:`, null不受影响；若带 `:`, 则连null值也受影响。

8. 计算shell字符串变量的长度：`$`

```
${#file}  #其值为27, 因为/dir1/dir2/dir3/my.file.txt刚好为27个字符。
```

9. bash数组(array)的处理方法

接下来，为大家介绍一下bash的数组(array)的处理方法。 一般而言, `A="a b c def"` 这样的变量只是将 `$A` 替换为一个字符串， 但是改为 `A=(a b c def)` , 则是将 `$A` 定义为数组....

1). 数组替换方法可参考如下方法：

```
${A[@]} #方法一
${A[*]} #方法二
```

以上两种方法均可以得到：a b c def, 即数组的全部元素。

2). 访问数组的成员**:**

```
${A[0]}
```

其中，`${A[0]}` 可得到a, 即数组A的第一个元素， 而 `${A[1]}` 则为数组A的第二元素，依次类推。

3). 数组的**length**：

```
${#A[@]} #方法一
${#A[*]} #方法二
```

以上两种方法均可以得到数组的长度: 4, 即数组的所有元素的个数。

回忆一下，针对字符串的长度计算，使用 `${#str_var}`；我们同样可以将该方法应用于数组的成员:

```
${#A[0]}
```

其中， `${#A[0]}` 可以得到：1，即数组A的第一个元素(a)的长度; 同理， `${#A[3]}` 可以得到: 3, 即数组A的第4个元素(def)的长度。

4). 数组元素的重新赋值：

```
A[3]=xyz
```

将数组A的第四个元素重新定义为xyz。

> **Tips:**
>
> 诸如此类的...
>
> 能够善用bash的$()与${}可以大大提高及 简化shell在变量上的处理能力哦~~~^_^

10. $(())作用:

好了，最后为大家介绍 `$(())` 的用途吧： **`$(())`** 是用来作整数运算的。

在bash中， `$(())` 的整数运算符号大致有这些：

- +- * / #分别为"加、减、乘、除"。
- % #余数运算,(模数运算)
- & | ^ ! #分别为"AND、OR、XOR、NOT"运算。

例如：

```
$ a=5; b=7; c=2;
$ echo $(( a + b * c ))
19
$ echo $(( (a + b)/c ))
6
$ echo $(( (a * b) % c ))
1
```

在 `$(())` 中的变量名称, 可以在其前面加 `$` 符号来替换， 也可以不用，如： `$(( $a + $b * $c ))` 也可以得到19的结果。

此外， `$(())` 还可作不同进制(如二进制、八进制、十六进制)的运算， 只是输出结果均为十进制的。

```
echo $(( 16#2a )) #输出结果为：42，(16进制的2a)
```

以一个实用的例子来看看吧：假如当前的umask是022,那么新建文件的权限即为：

```
$ umask 022
$ echo "obase=8; $(( 8#666 & (8#777 ^ 8#$(umask)) ))" | bc
644
```

事实上，单纯用 `(())` 也可以重定义变量值，或作testing：

```
a=5; ((a++)) #可将$a 重定义为6
a=5; ((a--)) #可将$a 重定义为4
a=5; b=7; ((a< b)) #会得到0 (true)返回值。
```

常见的用于 `(())` 的测试符号有如下这些：

符号	符号名称
<	小于号
>	大于号
<=	小于或等于
>=	大于或等于
==	等于
!=	不等于

> **Note:**
>
> 使用 `(())` 作整数测试时， 请不要跟 `[]` 的整数测试搞混乱了。
>
> 更多的测试，我们将于第10章为大家介绍。

怎样？ 好玩吧... ^_^

okay,这次暂时说这么多...

上面的介绍，并没有详列每一种可用的状态， 更多的，就请读者参考手册文件(man)吧...

shell十三问之9：$@与$*差在哪？

要说$@与$*之前， 需得先从shell script的positional parameter谈起...

我们都已经知道变量(variable)是如何定义和替换的， 这个不再多讲了。

1. shell script的positional parameter

但是，我们还需要知道有些变量是shell内定的， 且其名称是我们不能随意修改的。 其中，就有positional parameter在内。

在shell script中，我们可用$0, $1, $2, $3 ... 这样的变量分别提取命令行中的如下部分:

```
script_name parameter1 parameter2 parameter3 ...
```

我们很容易就能猜出, `$0` 就是代表 shell script名称(路径)本身， 而 `$1` 就是其后的第一个参数，如此类推...

须得留意的是 `IFS` 的作用, 也就是 `IFS` 被quoting处理后， 那么positional parameter也会改变。

如下例：

```
my.sh p1 "p2 p3" p4
```

由于p2与p3之间的空白键被soft quoting所关闭了， 因此，my.sh的中$2是"p2 p3",而$3则是p4...

还记得前两章，我们提到function时， 我们不是说过，它是script中的script吗？^_^

是的，function一样可以读取自己的(有别于script的) positional parameter, 唯一例外的是$0而已。

举例而言： 假设my.sh里有一个函数(function)叫my_fun, 若在script中跑 `my_fun fp1 fp2 fp3` , 那么，function内的$0就是my.sh，而$1是fp1而不是p1了...

不如写个简单的my.sh script 看看吧：

```
#!/bin/bash

my_fun() {
    echo '$0 inside function is '$0
    echo '$1 inside function is '$1
    echo '$2 inside function is '$2
}

echo '$0 outside function is '$0
echo '$1 outside function is '$1
echo '$2 outside function is '$2

my_fun fp1 "fp2 fp3"
```

然后在command line中跑一下 script就知道了：

```
chmod 755 my.sh

./my.sh p1 "p2 p3"
$0 outside function is ./my.sh
$1 outside function is p1
$2 outside function is p2 p3
$0 inside function is ./my.sh
$1 inside function is fp1
$2 inside function is fp2 fp3
```

然而，在使用positional parameter的时候， 我们要注意一些陷阱哦：

$10不是替换第**10**个参数， 而是替换第一个参数，然后在补一个**0**于其后;

也就是说， `my.sh one two three four five six seven eight nine ten` 这样的command line, my.sh里的$10不是ten而是one0 哦...小心小心 要抓到ten的话，有两种方法：

- 方法一：使用我们上一章介绍的${}, 也就是用${10}即可。
- 方法二：就是shift了。

用通俗的说法来说， 所谓的**shift**就是取消**positional parameter**中最左边的参数(**$0**不受影响)。 其预设值为1，也就是shift 或shift 1 都是取消$1, 而原本的$2则变成$1, $3则变成$2... 那亲爱的读者，你说要shift掉多少个参数， 才可用$1取得到${10} 呢？ ^_^

okay，当我们对positional parameter有了基本的概念之后， 那再让我们看看其他相关变量吧。

2. shell script的positional parameter的number

先是$#, 它可抓出positional parameter的数量。 以前面的 `my.sh p1 "p2 p3"` 为例： 由于"p2 p3"之间的 `IFS` 是在soft quote中， 因此，$#就可得到的值是2. 但如果p2与p3没有置于quoting中话， 那$#就可得到3的值了。 同样的规则，在function中也是一样。

因此，我们常在shell script里用如下方法， 测试script是否有读进参数：

```
[ $# = 0 ]
```

假如为0, 那就表示script没有参数，否则就是带有参数...

3. shell script中的$@与$*

接下来就是**$@**与**$*:** 精确来讲，两者只有在**soft quote**中才有差异， 否则，都表示**“**全部参数**” ($0**除外**)**。

若在comamnd line上， 跑 `my.sh p1 "p2 p3" p4` 的话， 不管$@还是$*, 都可得到 p1 p2 p3 p4 就是了。

但是，如果置于**soft quote**中的话：

- **"$@"**则可得到 **"p1" "p2 p3" "p4"** 这三个不同字段**(word);**
- **"$*"**则可得到 **"p1 p2 p3 p4"** 这一整个单一的字段。

我们修改一下前面的my.sh，使之内容如下：

```
#!/bin/bash

my_fun() {
    echo "$#"
}

echo 'the number of parameter in "$@" is ' $(my_fun "$@")
echo 'the number of parameter in "$*" is ' $(my_fun "$*")
```

然后再执行:

```
./my.sh p1 "p2 p3" p4
```

就知道，$@与$*差在哪了... ^_^

shell十三问之10：&& 与 || 差在哪？

好不容易，进入了两位数的章节了... 一路走来，很辛苦吧？也很快乐吧？ ^_^

在解答本章题目之前，先让我们了解一个概念： return value。

我们在shell下跑的每一个command或function， 在结束的时候都会传回父进程一个值，称为 `return value` 。

在shell command line中可用 `$?` ， 这个变量得到最"新"的一个 `return value` ， 也就是刚刚结束的那个进程传回的值。

`Return Value` (RV)的取值为0-255之间， 由进程或者script的作者自行定义：

- 若在script里，用exit RV 来指定其值; 若没有指定, 在结束时，以最后一个命令的RV，为script的RV值。
- 若在function里，则用return RV 来代替exit RV即可。

`Return Value` 的作用：用来判断进程的退出状态**(exit status)**. 进程的退出状态有两种：

- 0值为"真"(true)
- 非0值为"假"(false)

举个例子来说明好了： 假设当前目录内有一个my.file的文件， 而no.file是不存在的：

```
$ touch my.file
$ ls my.file
$ echo $? #first echo
0
$ ls no.file
ls: no.file: No such file or directory
$ echo $?     #second echo
1
$ echo $?     #third echo
0
```

上例的：

- 第一个echo是关于 `ls my.file` 的RV，可得到0的值，因此为true。
- 第二个echo是关于 `ls no.file` 的RV，得到非0的值，因此为false。
- 第三个echo是关于 `echo $?` 的RV，得到0值， 因此为true。

请记住： 每一个command在结束时，都会返回 `return value` ，不管你跑什么命令... 然而，有一个命令却是“专门”用来测试某一条而返回 `return value` ， 以供true或false的判断， 它就是 `test` 命令。

若你用的是bash， 请在command line下， 打 `man test` ，或者 `man bash` 来了解这个 `test` 的用法。 这是你可用作参考的最精准的文件了，要是听别人说的，仅作参考就好...

下面，我只简单作一些辅助说明，其余的一律以 `man` 为准： 首先， `test` 的表达式，我们称为expression，其命令格式有两种：

```
test expression
```

或者

```
[ expression ]
```

> **Note:**
>
> 请务必注意 `[]` 之间的空白键!

用哪一种格式无所谓，都是一样的效果。 (我个人比较喜欢后者...)

其次，bash的 `test` 目前支持的测试对象只有三种：

- string：字符串，也就是纯文字。
- integer：整数(0或正整数、不含负数或小数)
- file: 文件

请初学者，一定要搞清楚这三者的差异， 因为 `test` 所使用的expression是不一样的。

以A=123这个变量为例：

- `[ "$A" = 123 ]` #是字符串测试，测试$A是不是1、2、3这三个字符。
- `[ "$A" -eq 123 ]` #是整数测试，以测试$A是否等于123.
- `[-e "$A" ]` #文件测试，测试123这份文件是否存在.

第三， 当expression测试为“眞”时， `test` 就返回0(true)的 `return value` ; 否则，返回非0(false).

若在 expression 之前加一个 `!` (感叹号)，则在expression为假时，return value为0, 否则, return value 为非0值。

同时， `test` 也允许多重复合测试：

- expression1 -a expression2 #当两个expression都为true，返回0，否则，返回非0；
- expression1 -o expression2 #当两个expression均为false时，返回非0，否则，返回0；

例如：

```
[ -d "$file"  -a  -x "$file" ]
```

表示当$file是一个目录，且同时具有x权限时，test才会为true。

第四，在command line中使用 `test` 时，请别忘记命令行的“重组”特性， 也就是在碰到meta时，会先处理meta，在重新组建命令行。 (这个概念在第2章和第4章进行了反复强调)

比方说， 若 `test` 碰到变量或者命令替换时， 若不能满足 expression的格式时，将会得到语法错误的结果。

举例来说好了：

关于 `[ string1 = string2 ]` 这个test格式， 在等号两边必须要有字符串，其中包括空串(null串,可用soft quote或者hard quote取得)。

假如$A目前没有定义，或被定义为空字符串的话， 那如下的用法将会失败：

```
$ unset A
$ [ $A = abc ]
[: =: unary oprator expected
```

这是因为命令行碰到$这个meta时，会替换$A的值， 然后，再重组命令行，那就变成了： `[ = abc ]`, 如此一来，=的左边就没有字符串存在了， 因此，造成test的语法错误。 但是，下面这个写法则是成立的。

```
$ [ "$A" = abc ]
$ echo $?
1
```

这是因为命令行重组后的结果为： `[ "" = abc ]`, 由于等号的左边我们用soft quote得到一个空串， 而让test的语法得以通过...

读者诸君，请务必留意这些细节哦， 因为稍一不慎，将会导致 `test` 的结果变了个样。 若您对 `test` 还不是很有经验的话， 那在使用test时，不妨先采用如下这一个"法则":

若在 **`test`** 中碰到变量替换，用**soft quote**是最保险的*。

若你对quoting不熟的话，请重新温习第四章的内容吧...^_^

okay, 关于更多的 `test` 的用法，老话一句：请看其man page (`man test`)吧！^_^

虽然洋洋洒洒读了一大堆，或许你还在嘀咕...那...那个 `return value` 有啥用？

问得好: 告诉你：return value的作用可大了， 若你想要你的shell变"聪明"的话，就全靠它了：有了return value， 我们可以让shell根据不同的状态做不同的事情...

这时候，才让我来揭晓本章的答案吧~~~~^_^

`&&` 与 `||` 都是用来"组建" 多个command line用的；

- `command1 && command2` # command2只有在command1的RV为0(true)的条件下执行。
- `command1 || command2` # command2 只有在command1的RV为非0(false)的条件下执行。

以例子来说好了：

```
$ A=123
$ [ -n "$A" ] && echo "yes! it's true."
yes! it's true.
$ unset A
$ [ -n "$A" ] && echo "yes! it's true."
$ [ -n "$A" ] || echo "no, it's Not true."
no, it's Not true
```

> **Note:**
>
> `[ -n string ]`是测试string长度大于0, 则为true。

上例中，第一个`&&`命令之所以会执行其右边的`echo`命令， 是因为上一个`test`返回了0的RV值； 但第二个，就不会执行，因为`test`返回了非0的结果... 同理，`||`右边的`echo`会被执行，却正是因为左边的`test`返回非0所引起的。

事实上，我们在同一个命令行中，可用多个`&&` 或 `||` 来组建呢。

```
$ A=123
$ [ -n "$A" ] && echo "yes! it's true." || echo "no, it's Not ture."
yes! it's true.
$ unset A
$ [ -n "$A" ] && echo "yes! it's true." || echo "no, it's Not ture."
no, it's Not true
```

怎样，从这一刻开始，你是否觉得我们的shell是“很聪明”的呢？ ^_^

好了，最后布置一道练习题给大家做做看： 下面的判断是：当$A被赋值时，在看看其是否小于100，否则输出too big！

```
$ A=123
$ [ -n "$A" ] && [ "$A" -lt 100 ] || echo 'too big!'
$ too big!
```

若我取消A，照理说，应该不会输出文字啊，(因为第一个条件不成立)。

```
$ unset A
$ [ -n "$A" ] && [ "$A" -lt 100 ] || echo 'too big!'
$ too big!
```

为何上面的结果也可得到呢？ 又如何解决呢？

> **Tips:**
>
> 修改的方法有很多种， 其中一种方法可以利用第7章中介绍过 `command group`...

快告诉我答案，其余免谈....

解决方法1： `sub-shell` ：

```
$ unset A
$ [ -n "$A" ] && ( [ "$A" -lt 100 ] || echo 'too big!' )
```

解决方法二： `command group`：

```
$ unset A
$ [ -n "$A" ] && { [ "$A" -lt 100 ] || echo 'too big!'}
```

shell十三问之11：>与< 差在哪？

这次的题目，之前我在CU的shell版说明过了：(原帖的连接在论坛改版后，已经失效) 这次我就不重写了，将帖子的内容“抄”下来就是了...

1. 文件描述符(fd, File Descriptor)

谈到 `I/O redirection` ,不妨先让我们认识一下 `File Descriptor` (`fd` ，文件描述符)。

进程的运算，在大部分情况下，都是进行数据(data)的处理， 这些数据从哪里，读进来？又输出到哪里呢？ 这就是file descriptor(fd)的功用了。

在shell的进程中，最常使用的 `fd` 大概有三个，分别为:

- 0：standard Input (`STDIN`)
- 1: standard output(`STDOUT`)
- 2: standard Error output （ `STDERR` ）

在标准情况下，这些fd分别跟如下设备(device)关联：

- `stdin` (0): keyboard
- `stdout` (1): monitor
- `stderr` (2): monitor

> **Tips:** linux中的文件描述符(fd)用整数表示。 linux中任何一个进程都默认打开三个文件, 这三个文件对应的文件描述符分别是：0, 1, 2; 即stdin, stdout, stderr.

我们可以用如下命令测试一下：

```
$ mail -s test root
this is a test mail。
please skip.
^d (同时按下ctrl 跟d键)
```

很明显， `mail` 进程所读进的数据，就是从 `stdin` 也就是keyboard读进的。 不过，不见得每个进程的 `stdin` 都跟 `mail` 一样 从 `keyboard` 读进，因为进程的作者可以从文件参数读进 `stdin` ， 如：

```
$ cat /etc/passwd
```

但，要是 `cat` 之后没有文件参数则如何呢？ 哦， 请你自己玩玩看...^_^

```
$ cat
```

> **Tips:**
>
> 请留意数据输出到哪里去了， 最后别忘了按`ctrl+d`(`^d`), 退出stdin输入。

至于`stdout`与`stderr`，嗯...等我有空再续吧...^_^ 还是，有哪位前辈来玩接龙呢？

相信，经过上一个练习后， 你对`stdin`与`stdout`应该不难理解了吧？ 然后，让我们看看`stderr`好了。

事实上，`stderr`没什么难理解的： 说白了就是“错误信息”要往哪里输出而已... 比方说, 若读进的文件参数不存在的， 那我们在monitor上就看到了：

```
$ ls no.such.file
ls: no.such.file: No such file or directory
```

若同一个命令，同时成生`stdout`与`stderr`呢？ 那还不简单，都送到monitor来就好了：

```
$ touch my.file
$ ls my.file on.such.file
ls: no.such.file: No such file or directory
my.file
```

okay, 至此，关于fd及其名称、还有相关联的设备， 相信你已经没问题了吧？

---

2. I/O 重定向(I/O Redirection)

那好，接下来让我们看看如何改变这些fd的预设数据通道。

- 用`<`来改变读进的数据通道(stdin),使之从指定的文件读进。
- 用`>`来改变输出的数据通道(stdout，stderr),使之输出到指定的文件。

2.1 输入重定向 `n<` (input redirection)

比方说：

```
$ cat < my.file
```

就是从my.file读入数据

```
$ mail -s test root < /etc/passwd
```

则是从/etc/passwd读入...

这样一来，stdin将不再是从keyboard读入， 而是从指定的文件读入了...

严格来说， `<` 符号之前需要指定一个**fd**的**(**之前不能有空白**)**，但因为**0**是 `<` 的预设值，因此， `<` 与 `0<` 是一样的*。

okay，这样好理解了吧？

那要是用两个 `<` ，即 `<<` 又是啥呢？ 这是所谓的 `here document` **,** 它可以让我们输入一段文本， 直到读到 `<<` 后指定的字符串。

比方说：

```
$ cat <
```

这样的话, `cat` 会读入3个句子， 而无需从keyboard读进数据且要等到(ctrl+d, ^d)结束输入。

2.2 重定向输出 `>n` (output redirection)

当你搞懂了 `0<` 原来就是改变 `stdin` 的数据输入通道之后， 相信要理解如下两个redirection就不难了：

- `1>` #改变stdout的输出通道；
- `2>` #改变stderr的输出通道；

两者都是将原来输出到monitor的数据， 重定向输出到指定的文件了。

由于**1**是 `>` 的预设值， 因此， `1>` 与 `>` 是相同的，都是改变 `stdout` .

用上次的ls的例子说明一下好了:

```
$ ls my.file no.such.file 1>file.out
ls: no.such.file: No such file or directory
```

这样monitor的输出就只剩下 `stderr` 的输出了， 因为 `stdout` 重定向输出到文件file.out去了。

```
$ ls my.file no.such.file 2>file.err
my.file
```

这样monitor就只剩下了 `stdout` , 因为 `stderr` 重定向输出到文件file.err了。

```
$ ls my.file no.such.file 1>file.out 2>file.err
```

这样monitor就啥也没有了， 因为 `stdout` 与 `stderr` 都重定向输出到文件了。

呵呵，看来要理解 `>` 一点也不难啦是不？ 没骗你吧？ ^_^ 不过有些地方还是要注意一下的。

```
$ ls my.file no.such.file 1>file.both 2>file.both
```

假如 `stdout` (1)与 `stderr` (2)都同时在写入file.both的话， 则是采取"覆盖"的方式：后来写入覆盖前面的。

让我们假设一个 `stdout` 与 `stderr` 同时写入到file.out的情形好了；

- 首先 `stdout` 写入10个字符
- 然后 `stderr` 写入6个字符

那么，这时原本的 `stdout` 输出的10个字符， 将被 `stderr` 输出的6个字符覆盖掉了。

那如何解决呢？所谓山不转路转，路不转人转嘛， 我们可以换一个思维： 将 `stderr` 导进 `stdout` 或者将 `stdout` 导进到 `stderr` , 而不是大家在抢同一份文件，不就行了。 bingo就是这样啦：

- 2>&1 #将 `stderr` 并进 `stdout` 输出
- 1>&2 或者 >&2 #将 `stdout` 并进 `stderr` 输出。

于是，前面的错误操作可以改写为:

```
$ ls my.file no.such.file 1>file.both 2>&1
$ ls my.file no.such.file 2>file.both >&2
```

这样，不就皆大欢喜了吗？ ~~~ ^_^

不过，光解决了同时写入的问题还不够， 我们还有其他技巧需要了解的。 故事还没有结束，别走开广告后，我们在回来....

2.3 I/O重定向与linux中的 `/dev/null`

---

okay，这次不讲I/O Redirection, 请佛吧... (有没有搞错？ `网中人` 是否头壳烧坏了？...)嘻~~~^_^

学佛的最高境界，就是"四大皆空"。 至于是空哪四大块，我也不知，因为我还没有到那个境界.. 这个“空”字,却非常值得反复把玩： ---色即是空，空即是色 好了，施主要是能够领会"空"的禅意，那离修成正果不远了。

在linux的文件系统中，有个设备文件: `/dev/null` . 许多人都问过我，那是什么玩意儿？ 我跟你说好了，那就是"空"啦。

没错空空如也的空就是null了... 请问施主是否忽然有所顿悟了呢？ 然则恭喜了。

这个null在 I/O Redirection中可有用的很呢？

- 将fd `1` 跟fd `2` 重定向到/dev/null去，就可忽略stdout, stderr的输出。
- 将fd `0` 重定向到/dev/null，那就是读进空(nothing).

比方说，我们在执行一个进程时，会同时输出到stdout与stderr， 假如你不想看到stderr(也不想存到文件)， 那就可以：

```
$ ls my.file no.such.file 2>/dev/null
my.file
```

若要相反：只想看到stderr呢？ 还不简单将stdout，重定向的/dev/null就行：

```
$ ls my.file no.such.file >/dev/null
ls: no.such.file: No such file or directory
```

那接下来，假如单纯的只跑进程，而不想看到任何输出呢？ 哦，这里留了一手，上次没讲的法子,专门赠与有缘人... ^_^ 除了用 `>/dev/null 2>&1` 之外，你还可以如此：

```
$ ls my.file no.such.file &>/dev/null
```

> **Tips:**
>
> 将&>换成>&也行！

2.4 重定向输出append (`>>`)

---

okay？ 请完佛，接下来，再让我们看看如下情况：

```
$ echo "1" > file.out
$ cat file.out
1
$ echo "2" > file.out
$ cat file.out
2
```

看来，我们在重定向stdout或stderr进一个文件时， 似乎永远只能获得最后一次的重定向的结果. 那之前的内容呢？

呵呵，要解决这个问题，很简单啦，将 `>` 换成 `>>` 就好了；

```
$ echo "3" >> file.out
$ cat file.out
2
3
```

如此一来，被重定向的文件的之前的内容并不会丢失， 而新的内容则一直追加在最后面去。so easy?...

但是，只要你再次使用 `>` 来重定向输出的话， 那么，原来文件的内容被truncated(清洗掉)。这是，你要如何避免呢？ ----备份， yes，我听到了，不过，还有更好的吗？ 既然与施主这么有缘分，老衲就送你一个锦囊妙法吧：

```
$ set -o noclobber
$ echo "4" > file.out
-bash：file: cannot overwrite existing file.
```

那，要如何取消这个限制呢? 哦，将 `set -o` 换成 `set +o` 就行了：

```
$ set +o noclobber
$ echo "5" > file.out
$ cat file.out
5
```

再问：那有办法不取消而又“临时”改写目标文件吗？ 哦，佛曰：不可告也。 啊，~~~开玩笑的，开玩笑啦~~~^_^， 哎，早就料到人心是不足的了

```
$ set -o noclobber
$ echo "6" >| file.out
$ cat file.out
6
```

留意到没有： 在 `>` 后面加个 `|` 就好， 注意： `>` 与 `|` 之间不能有空白哦...

2.5 I/O Redirection的优先级

呼....(深呼吸吐纳一下吧)~~~ ^_^ 再来还有一个难题要你去参透呢:

```
$ echo "some text here" >file
$ cat < file
some text here
$cat < file >file.bak
$cat < file.bak
some text here
$cat < file >file
```

嗯？注意到没有？ ---怎么最后那个cat命令看到file是空的呢？ why？ why？ why？

前面提到： `$cat < file > file` 之后， 原本有内容的文件，结果却被清空了。 要理解这个现象其实不难， 这只是priority的问题而已： 在**IO Redirection**中, **stdout**与**stderr**的管道先准备好， 才会从**stdin**读入数据。 也就是说，在上例中， `>file` 会将file清空， 然后才读入 `< file` 。 但这时候文件的内容已被清空了，因此就变成了读不进任何数据。

哦，~~原来如此~~~^_^ 那...如下两例又如何呢？

```
$ cat <> file
$ cat < file >>file
```

嗯...同学们，这两个答案就当练习题喽， 下课前交作业。

> **Tips:** 我们了解到 `>file` 能够快速把文件file清空； 或者使用 `:>file` 同样可以清空文件， `:>file` 与 `>file` 的功能： 若文件file存在，则将file清空; 否则，创建空文件file (等效于 `touch file`); 二者的差别在于 `>file` 的方式不一定在所有的shell的都可用。
>
> `exec 5<>file; echo "abcd" >&5; cat <&5` 将file文件的输入、输出定向到文件描述符5，从而描述符5可以接管file的输入输出； 因此， `cat <>file` 等价于 `cat < file` 。
>
> 而 `cat < file >>file` 则使file内容成几何级数增长。

好了， I/O Redirection也快讲完了， sorry,因为我也只知道这么多而已啦~~嘻~~~^_^ 不过，还有一样东东是一定要讲的，各位观众(请自行配乐~!#@$%): 就是 `pipe line` 也。

2.6 管道(pipe line)

谈到 `pipe line` ，我相信不少人都不会陌生： 我们在很多command line上常看到 `|` 符号就是pipe line了。

不过，pipe line究竟是什么东东呢？ 别急别急...先查一下英文字典，看看pipe是什么意思？没错他就是“水管”的意思... 那么，你能想象一下水管是怎样一个根接一根的吗？ 又， 每根水管之间的input跟output又如何呢？ 灵光一闪：原来pipe line的I/O跟水管的I/O是一模一样的：上一个命令的**stdout**接到下一个命令的**stdin**去了 的确如此。不管在command line上使用了多少个pipe line， 前后两个command的I/O是彼此连接的 (恭喜：你终于开放了 ^_^)

不过...然而...但是... ...stderr呢？好问题不过也容易理解：若水管漏水怎么办？也就是说：在pipe line之间, 前一个命令的stderr是不会接进下一个命令的stdin的， 其输出，若不用2>file的话，其输出在monitor上来。 这点请你在pipe line运用上务必要注意的。

那，或许你有会问: 有办法将**stderr**也喂进下一个命令的**stdin**吗？ (贪得无厌的家伙)，方法当然是有的，而且，你早已学习过了。 提示一下就好：**请问你如何将stderr合并进stdout一同输出呢？ 若你答不出来，下课后再来问我...(如果你脸皮足够厚的话...)

或许，你仍意犹未尽，或许，你曾经碰到过下面的问题： 在 `cmd1 | cmd2 | cmd3 | ...` 这段pipe line中如何将cmd2的输出保存到一个文件呢？

若你写成 `cmd1 | cmd2 >file | cmd3` 的话， 那你肯定会发现 `cmd3` 的stdin是空的，(当然了，你都将 水管接到别的水池了) 聪明的你或许会如此解决：

```
cmd1 | cmd2 >file; cmd3 < file
```

是的，你可以这样做，但最大的坏处是： file I/O会变双倍，在command执行的整个过程中，file I/O是最常见的最大效能杀手。 凡是有经验的shell操作者，都会尽量避免或降低file I/O的频度。

那上面问题还有更好的方法吗？ 有的，那就是 `tee` 命令了。 所谓的 **`tee`** 命令是在不影响原本**I/O**的情况下， 将**stdout**赋值到一个文件中去。 因此，上面的命令行，可以如此执行：

```
cmd1 | cmd2 | tee file | cmd3
```

在预设上， `tee` 会改写目标文件， 若你要改为追加内容的话，那可用-a参数选项。

基本上，pipe line的应用在shell操作上是非常广泛的。 尤其是在text filtering方面， 如，cat, more, head, tail, wc, expand, tr, grep, sed, awk...等等文字处理工具。 搭配起pipe line 来使用，你会觉得 command line 原来活得如此精彩的。 常让人有“众里寻他千百度，蓦然回首，那人却在灯火阑珊处”之感...

好了，关于I/O Redirection的介绍就到此告一段落。 若日后，有空的话，在为大家介绍其他在shell上好玩的东西。

shell十三问之12：你要if还是case呢？

还记得我们在第10章所介绍的 `return value` 吗？

是的，接下来的介绍的内容与之有关， 若你的记忆也被假期所抵消的话， 那建议您还是回去温习温习再回来...

若你记得 `return value` ，我想你也应该记得了 `&&` 与 `||` 什么意思吧? 用这两个符号再搭配command group的话， 我们可让shell script变得更加聪明哦。 比方说：

```
cmd1 && {
    cmd2
    cmd3
    ;
} || {
    cmd4
    cmd5
}
```

意思是说： 若 cmd1的 `return value` 为true的话， 然后执行cmd2与cmd3， 否则执行cmd4与cmd5.

事实上， 我们在写shell script的时候， 经常需要用到这样、那样的条件 以作出不同的处理动作。 用 `&&` 与 `||` 的确可以达成条件执行的结果， 然而，从“人类语言”上来理解， 却不是那么直观。 更多时候，我们还是喜欢用 `if...then...else...` 这样的的keyword来表达条件执行。

在bash shell中，我们可以如此修改上一段代码：

```
if cmd1
then
    cmd2
    cmd3
else
    cmd4
    cmd5
fi
```

这也是我们在shell script中最常用的 `if` 判断式： 只要 `if` 后面的command line返回true的return value (我们常用 `test` 命令返回的return value)， 然则就执行 `then` 后面的命令，否则，执行 `else` 之后的命令， `fi` 则是用来结束判断式的keyword。

在 `if` 的判断式中， `else` 部分可以不用，但 `then` 是必需的。 (若 `then` 后不想跑任何command，可用 `:` 这个 `null command` 代替)。 当然，then或else后面，也可以再使用更进一层的条件判断式， 这在shell script的设计上很常见。 若有多项条件需要"依序"进行判断的话， 那我们则可使用 `elif` 这样的keyword：

```
if cmd1; then
    cmd2;
elif cmd3; then
     cmd4
else
     cmd5
fi
```

意思是说： 若cmd1为true，然则执行cmd2； 否则在测试cmd3，若为true则执行cmd4； 倘若cmd1与cmd3均不成立，那就执行cmd5。

`if` 判断式的例子很常见，你可从很多shell script中 看得到，我这里不再举例子了...

接下来为要为大家介绍的是 `case` 判断式。 虽然 `if` 判断式已可应付大部分的条件执行了，然而，在某些场合中，却不够灵活， 尤其是在**string**式样的判断上，比方如下：

```
QQ() {
    echo -n "Do you want to continue? (Yes/No): "
     read YN
     if [ "$YN" = Y -o "$YN" = y -o "$YN" = "Yes" -o "$YN" = "yes" -o "$YN" = YES]
     then
         QQ
     else
         exit 0
     fi
}

QQ
```

从例中，我们看得出来， 最麻烦的部分是在判断YN的值可能有好几种样式。

聪明的你或许会如此修改：

```
 QQ() {
     echo -n "Do you want to continue? (Yes/No): "
     read YN
     if echo "$YN" | grep -q '^[Yy]\([Ee][Ss]\)*$'
     then
         QQ
     else
         exit 0
     fi
}

QQ
```

也就是用 `Regular Expression` 来简化代码。(我们有机会，再来介绍 `RE`) 只是...是否有其他更方便的方法呢？ 有的，就是用 `case` 判断式即可：

```
 QQ() {
     echo -n "Do you want to continue? (Yes/No): "
     read YN
     case "$YN" in
         [Yy]|[Yy][Ee][Ss])
             QQ
             ;;
         *)
             exit 0
             ;;
     esac
}

QQ
```

我们常用的 `case` 的判断式来判断某一变量 在不同的值(通常是string)时，作出不同的处理，比方说， 判断**script**参数，以执行不同的命令。

若你有兴趣，且用linux系统的话， 不妨挖一挖 `/etc/init.d/*` 中的那堆script中的 `case` 用法. 如下就是一例：

```
case "$1" in
    start)
        start
        ;;
    stop)
        stop
        ;;
    status)
        rhstatus
        ;;
    restart|reload)
        restart
        ;;
    condrestart)
        [ -f /var/lock/subsys/syslog ] && restart || :
        ;;

    *)
        echo $"Usage: $0 {start|stop|status|restart|condrestart}"
        exit 1
esac
```

(若你对 postional parameter的印象已经模糊了，请重看第9章吧。)

okay，是十三问还剩一问而已，过几天再来搞定之...^_^

shell十三问之13： for what？ while与until差在哪？

终于，来到了shell十三问的最后一问了... 长长吐一口气~~~~

最后要介绍的是shell script设计中常见的`循环`(`loop`). 所谓的`loop`就是script中的一段在一定条件下反复执行的代码。

bash shell中常用的`loop`有如下三种：

- for
- while
- until

1. for loop

`for` loop 是从一个清单列表中读进变量的值， 并依次的循环执行`do`到`done`之间的命令行。 例：

```
for var in one two three four five
do
    echo -----------------
    echo '$var is '$var
    echo
done
```

> 上例的执行结果将会是：
>
> 1. for会定义一个叫var的变量，其值依次是one two three four five。
>   i. 因为有5个变量值，因此，`do`与`done`之间的命令行会被循环执行5次。
>   ii. 每次循环均用`echo`产生3个句子。而第二行中不在hard quote之内的$var会被替换。
>   iii. 当最后一个变量值处理完毕，循环结束。

我们不难看出，在`for` loop中，变量值的多寡，决定循环的次数。 然而，变量在循环中是否使用则不一定，得视设计需求而定。 倘若`for` loop没有使用in这个keyword来制变量清单的话，其值将从`$@`(或`$*`)中继承：

```
for var; do
    ......
done
```

> **Tips:**
>
> 若你忘记了`positional parameter, 请温习第9章...

`for` loop用于处理“清单”(list)项目非常方便， 其清单除了明确指定或从`postional parameter`取得之外， 也可以从`变量替换`或者`命令替换`取得... (再一次提醒：别忘了命令行的“重组”特性) 然而，对于一些“累计变化”的项目(整数的加减)，for也能处理：

```
for ((i = 1; i
```

2. while loop

除了`for` loop, 上面的例子， 我们也可改用`while` loop来做到：

```
num=1
while [ "$num" -le 10 ]; do
    echo "num is $num"
    num=$(($num + 1))
done
```

`while` loop的原理与`for` loop稍有不同： 它不是逐次处理清单中的变量值， 而是取决于`while` 后面的命令行的return value：

- 若为true， 则执行`do`与`done`之间的命令， 然后重新判断`while`后的return value。
- 若为false，则不再执行`do`与`done`之间的命令而结束循环。

> 分析上例：
>
> 1. 在`while`之前，定义变量num=1.
>   i. 然后测试(`test`)$num是否小于或等于10.
> 2. 结果为true，于是执行`echo`并将num的值加1.
> 3. 再作第二轮测试，此时num的值为1+1=2，依然小于或等于10，因此，为true，循环继续。
> 4. 直到num为10+1=11时，测试才会失败...于是结束循环。

我们不难发现： 若 **`while`** 的测试结果永远为**true**的话，那循环将一直永久执行下去：

```
while:; do
    echo looping...
done
```

上面的`:`是**bash**的**null command**，不做任何动作， 除了返回**true**的**return value**。 因此这个循环不会结束，称作死循环。

死循环的产生有可能是故意设计的(如跑daemon)， 也可能是设计的错误。

若要结束死循环，可通过signal来终止(如按下ctrl-c). (关于process与signal，等日后有机会再补充，十三问略过。)

3.until loop

一旦你能够理解 `while` loop的话，那就能理解 `until` loop: **与 `while` 相反， `until` 是在return value 为false时进入循环，否则，结束。 因此，前面的例子，我们也可以轻松的用 `until` 来写：

```
num=1
until [ ! "$num" -le 10 ]; do
    echo "num is $num"
    num=$(($num + 1))
done
```

或者：

```
num=1

until [ "$num" -gt 10 ]; do
    echo "num is $num"
    num=$(($num + 1))
done
```

okay, 关于bash的三个常用的loop暂时介绍到这里。

4. shell loop中的break与continue

在结束本章之前，再跟大家补充两个loop有关的命令：

- `break`
- `continue` 这两个命令常用在复合式循环里， 也就是 `do ... done` 之间又有更进一层的loop， 当然，用在单一循环中也未尝不可啦... ^_^

`break` 用来中断循环，也就是强迫结束循环。 若 `break` 后面指定一个数值n的话，则从里向外中断第n个循环， 预设值为 `break 1` ，也就是中断当前循环。 在使用break时，需要注意的是，它与 `return` 及 `exit` 是不同的：

- `break` 是结束loop；
- `return` 是结束function；
- `exit` 是结束script/shell;

而 `continue` 则与 `break` 相反：强迫进入下一次循环动作.

若你理解不来的话，那你可简单的看成： 在 `continue` 在 `done` 之间的句子略过而返回到循环的顶端...

与 `break` 相同的是： `continue` 后面也可以指定一个数值n， 以决定继续哪一层(从里往外计算)的循环， 预设值为 `continue 1`,也就是继续当前的循环。

在shell script设计中，若能善用loop， 将能大幅度提高script在复杂条件下的处理能力。 请多加练习吧...

---

shell是十三问的总结语

好了，该是到了结束的时候了。 婆婆妈妈地跟大家啰嗦了一堆shell的基础概念。

目的不是要告诉大家“答案”，而是要带给大家“启发”...

在日后的关于shell的讨论中，我或许经常用"连接"的方式 指引十三问中的内容。

以便我们在进行技术探讨时，彼此能有一些讨论的基础， 而不至于各说各话、徒费时力。

但更希望十三问能带给你更多的思考与乐趣， 至为重要的是通过实践来加深理解。

是的，我很重视实践与独立思考这两项学习要素。

若你能够掌握其中的真谛，那请容我说声 ： 恭喜十三问你没白看了 ^_^

p.s. 至于补充问题部分，我暂时不写了。 而是希望：

1. 大家补充题目。
2. 一起来写心得。

Good luck and happy studing！

---

shell十三问原作者`网中人`签名中的bash的fork bomb

最后，Markdown整理者补上本书的原作者网中人的个性签名：

> 君子博学而日叁省乎己，则知明而行无过矣。
>
> 一个能让系统shell崩溃的shell 片段：

```
:() { :|:& }; :       #
```

原来是一个bash的fork炸弹：ref：http://en.wikipedia.org/wiki/Fork_bomb

整理后的代码：

```
:() {
    :|:&
}
:
```

> 代码分析：
>
> (即除最后一行外)
>
> 定义了一个 shell 函数，函数名是`:`，
>
> 而这个函数体执行一个后台命令`:|:`
>
> 即冒号命令(或函数，下文会解释)的输出 通过管道再传给冒号命令做输入
>
> 最后一行执行“:”命令

在各种shell中运行结果分析：

> 这个代码只有在 **bash** 中执行才会出现不断创建进程而耗尽系统资源的严重后果;
>
> 在 ksh (Korn shell), sh (Bourne shell)中并不会出现，
>
> 在 ksh88 和传统 unix Bourne shell 中冒号不能做函数名，
>
> 即便是在 unix-center freebsd 系统中的 sh 和 pdksh（ksh93 手边没有，没试）中冒号可以做函数名，但还是不会出现那个效果。
>
> 原因是 sh、ksh 中内置命令的优先级高于函数，所以执行“:”，总是执行内置命令“:”而不是刚才定义的那个恐怖函数。
>
> 但是在 **bash** 中就不一样，bash 中函数的优先级高于内置命令，所以执行“:”结果会导致不断的递归，而其中有管道操作，这就需要创建两个子进程来实现，这样就会不断的创建进程而导致资源耗尽。

众所周知，bash是一款极其强大的shell，提供了强大的交互与编程功能。

这样的一款shell中自然不会缺少“函数”这个元素来帮助程序进行模块化的高效开发与管理。于是产生了由于其特殊的特性，bash拥有了fork炸弹。

Jaromil在2002年设计了最为精简的一个fork炸弹的实现。

> 所谓fork炸弹是一种恶意程序，它的内部是一个不断在fork进程的无限循环.
>
> fork炸弹并不需要有特别的权限即可对系统造成破坏。
>
> fork炸弹实质是一个简单的递归程序。
>
> 由于程序是递归的，如果没有任何限制，
>
> 这会导致这个简单的程序迅速耗尽系统里面的所有资源.

shell十三问之14: 跟[!]差在哪？ (wildcard)

这个题目说穿了， 就是要探讨Wildcard与Regular Expression的差别的。 这也是很多初学shell的朋友很容易混淆的地方。

首先，让我们回到十三问之第2问， 再一次将我们提到的command line format 温习一次：

```
command_name options arguments
```

同时，也再来理解一下，我在第5章所提到的变量替换的特性：

```
先替换，再重组 command line!
```

有了这个两个基础后，再让我们来看Wildcard是什么回事吧。

Part-I Wildcard （通配符）

首先，

```
`Wildcard` 也是属于 `command line` 的处理工序，作用于 `arguments` 里的 `path` 之上。
```

没错，它不用在`command_name`，也不用在`options`上。 而且，若argument不是path的话，那也与wildcard无关。

换句更为精确的定义来讲，

```
 `wildcard`是一种命令行的路径扩展(path expansion)功能。
```

提到这个扩展，那就不要忘了 command line的“重组”特性了！

是的，这与`变量替换`(variable subtitution)及`命令替换`(command substitution)的重组特性是一样的。

也就是在`wildcard`进行扩展后， 命令行会先完成重组，才会交给shell来处理。

了解了`wildcard`的扩展与重组特性后， 接下来，让我们了解一些常见的wildcard吧。

wildcard	功能
*	匹配0个或多个字符
?	匹配任意单一字符
[list]	匹配list中任意单一字符
[!list]	匹配不在list中任意单一字符
{string1,string2,...}	匹配string1或者stsring2或者(...)中其一字符串

Note: list 中可以指定单个字符，如abcd, 也可以指定ASCII字符的起止范围，如 a-d。 即 [abcd] 与 [a-d] 是等价的，称为一个自定义的字符类。

例如：

```
a*b     # a 与 b 之间可以有任意个字符（0个或多个），如aabcb, axyzb, a012b,ab等。
a?b     # a 与 b 之间只能有一个字符，但该字符可以任意字符，如 aab, abb, acb, azb等。
a[xyz]b # a 与 b 之间只能有一个字符，但这个字符只能是x或者y或者z，如：axb, ayb, azb这三个。
a[!0-9]b# a 与 b 之间只能有一个字符，但这个字符不能是阿拉伯数字，如aab，ayb，a-b等。
a{abc,xyz,123}b # a 与 b之间只能是abc或者xyz或者123这三个字串之一，扩展后是aabcb，axyzb，a123b。
```

1. `[! ]` 中的 `!` 只有放在第一位时，才有取反的功效。 eg: `[!a]*` 表示当前目录下不以a开头的路径名称； `/tmp/[a\!]*` 表示/tmp目录下所有以a 或者 ! 开头的路径名称；

   思考：为何!前面要加\呢？提示是十三问之4.

2. `[ - ]` 中 `-` 左右两边均有字符时，才表示一个范围，否则,仅作 `-` (减号)字符来处理。 举例： `/tmp/*[-z]/[a-zA-Z]*` 表示/tmp 目录下所有以z或者-结尾的子目录中， 以英文字母(不分大小写)开头的目录名称。

3. 以*或?开头的wildcard不能匹配隐藏文件(即以.开头的文件名)。 eg: `*.txt` 并不能匹配 `.txt` 但能匹配1.txt这样的路径名。 但1*txt及1?txt均可匹配1.txt这样的路径名。

基本上，要掌握wildcard并不难， 只要多加练习，再勤于思考，就能灵活运用了。

再次提醒：

```
别忘了wildcard的"扩展" + "重组" 这个重要特性，而且只作用在 argument的path上。
```

比方说， 假如当前目录下有： a.txt b.txt c.txt 1.txt 2.txt 3.txt 这几个文件。

当我们在命令行中执行 `ls -l [0-9].txt` 的命令行时， 因为wildcard处于argument的位置上，

于是根据匹配的路径，扩展为: 1.txt 2.txt 3.txt， 在重组出 `ls -l 1.txt 2.txt 3.txt` 这样的命令行。

因此，你在命令行上敲 `ls -l [0-9].txt` 与 `ls -l 1.txt 2.txt 3.txt` 输出的结果是一样，原因就是在于此。

shell十三问之15: 跟[!]差在哪？ (RE: Regular Expression）

Part-II Regular Expression (正则表达式)

接下来的Regular Expression(RE) 可是个大题目，要讲的很多。 我这里当然不可能讲得很全。 只希望能带给大家一个基本的入门概念，就很足够了...

先来考一下英文好了：What is expression? 简单来说，就是"表达"，也就是人们在沟通的时候所要陈述的内容。

然而，生活中，表达方要清楚的将意思描述清楚， 而让接收方完整无误地领会，可不是件容易的事情。

因而才会出现那么多的"误会", 真可叹句"表达不易"啊......

同样的情形也发生在计算机的数据处理过程中， 尤其是当我们在描述一段"文字内容"的时候.... 那么，我们不禁要问： 有何方法可以让大家的误会降至最低程度， 而让表达的精确度达到最高程度呢？ 答案就是"标准化"了， 也就是我们这里要谈的 `Regular Expression` 啦...^_^

然而，在进入 `RE` 介绍之前，不妨先让我们温习一下shell十三问之第4问， 那就是关于quoting的部分。

关键是要能够区分 **shell command line**上的**meta**与**literal**的这两种不同的字符类型。

然后，我这里也跟你讲： **RE** 表达式里字符也分**meta**与**literal**这两种。

呵，不知亲爱的读者是否被我搞混乱了呢？... ^_^

这也难怪啦，因为这的确是最容易混淆的地方， 刚学 `RE` 的朋友很多时候，都死在这里！ 因此，请特别小心理解哦...

简单而言，除非你将 `RE` 写在特定程序使用的脚本里， 否则，我们的 `RE` 也是通过 command line输入的。 然而， 不少**RE**所使用的**meta**字符，跟**shell** 的**meta**字符是冲突的。

比方说， `*` 这个字符，在**RE**里是一个**modifier(**修饰符**);**而在**command line**上，确是**wildcard(**通配符**)**。

那么，我们该如何解决这样的冲突呢？ 关键就是看你对shell十三问的第4问中所提的quoting是否足够理解了！

若你明白到 **shell quoting** 就是用来在**command line**上关闭**shell meta**这一基本原理， 那你就能很轻松的解决 RE meta与shell meta的冲突问题了： 用**shell quoting** 关闭掉**shell meta**就是了。 就这么简单... ^_^

再以刚提到 `*` 字符为例， 若在command line的path中没有quoting处理的话， 如abc* 就会被作为wildcard expression来扩充及重组了。 若将其置于quoting中，即"abc*"，则可以避免wildcard expand的处理。

好了，说了大半天，还没有进入正式的RE介绍呢.... 大家别急，因为我的教学风格就是要先建立基础，循序渐进的... ^_^ 因此, 我这里还要再啰嗦一个观念，才会到RE的说明啦...(哈...别打我...)

当我们在谈到RE时，千万别跟wildcard搞混在一起！ 尤其是

```
在command line的位置里，wildcard只作用于argument的path上；
而RE却只用于"字符串处理" 的程序中，这与路径名一点关系也没有。
```

> **Tips:** RE 所处理的字符串，通常是指纯文本或通过stdin读进的内容。

okay，够了够了，我已看到一堆人开始出现不耐烦的样子了... ^_^ 现在，就让我们登堂入室，揭开RE的神秘面纱吧， 这样可以放过我了吧？ 哈哈...

在RE的表达式里，主要分为两种字符： `literal` 与 `meta` 。 所谓 `literal` 就是在RE里不具有特殊功能的字符，如abc，123等； 而 `meta` ,在RE里具有特殊的功能。 要关闭之，需要在 `meta` 之前使用escape()转义字符。

然而，在介绍 `meta` 之前，先让我们来认识一下字符组合(character set)会更好些。

一、所谓的char set就是将多个连续的字符作为一个集合。 例如：

char set	意义	
abc	表示abc三个连续的字符，但彼此独立而非集合。(可简单视为三个char set)	
(abc)	表示abc这三个连续字符的集合。(可简单视为一个char set)	
abc\	xyz	表示abc或xyz这两个char set之一
[abc]	表示单一字符，可为a或b或c;与wildcard的[abc]原理相同，称之为字符类。	
abc	表示单一字符，不为a或b或c即可。(与wildcard [!abc]原理相同)	
.	表示任意单个字符，(与wildcard的?原理相同)	

note: abc|xyz 表示abc或xyz这两个char set之一

在认识了RE的char set这个概念之后，然后，在让我们多认识几个RE中常见的meta字符：

二、 锚点(anchor): 用以标识RE在句子中的位置所在。 常见的有：

锚点	说明
^	表示句首。如，^abc表示以abc开头的句子。
$	表示句尾。如，abc$表示以abc结尾的句子。
\<	表示词首。如，\<abc表示以abc开头的词。
\>	表示词尾。如，abc\>表示以abc结尾的词。

三、 修饰符(modifier)：独立表示时本身不具意义，专门用以修饰前一个char set出现的次数。 常见的有：

modifier	说明
*	表示前一个char set出现0次或多次，即任意次。如ab*c表示a与c之间可以有0个或多个b。
?	表示前一个char set出现0次或1次，即至多出现1次。如ab?c 表示a与c之间可以有0个或1个b。
+	表示前一个char set出现1次或多次，即至少出现1次。如ab+c 表示a与c之间可以有1个或多个b。
{n}	表示前一个char set出现n次。如ab{n}c 表示a与c之间可以有n个b。
{n, }	表示前一个char set至少出现n次。如ab{n}c 表示a与c之间至少有n个b。
{n, m}	表示前一个char set至少出现n次，至多出现m次。如ab{n，m}c 表示a与c之间至少有n个b，至多有m个b。

然而，当我们在识别modifier时，却很容易忽略"边界(boundary)字符"的重要性。

以 `ab{3,5}c` 为例，这里的a与c就是边界字符了。 若没有边界字符的帮忙，我们很容易做出错误的解读。 比方说: 我们用 `ab{3,5}` 这个RE（少了c这个边界字符) 可以抓到"abbbbbbbbbb" (a后面有10个b)的字符串吗？ 从刚才的modifier的说明，我们一般认为，我们要的b是3到5个， 若超出了此范围，就不是我们所要表达的。 因此，我们或许会很轻率地认为这个RE抓不到结果（上述"abbbbbbbbbb"字符串）。

然而，答案却是可以的！为什么呢？ 让我们重新解读 `ab{3,5}` 这个RE看看： 我们要表达的是a后接3到5个b即可，但3到5个b后面，我们却没有规定什么， 因此，在RE后面可以是任意的字符串，当然包括b也可以啦！(明白了吗？)

同样，我们用 `b{3,5}c` 也同样可以抓到"abbbbbbbbbbc" 这样的字符串。

但当我们用 `ab{3,5}c` 这样的RE时， 由于同时有a与c这连个边界字符，就截然不同了！

有空在思考一下，为何我们用下面这些RE都抓到abc这样的字符串呢？

```
x*
ax*, abx*, ax*b
abcx*, abx*c, ax*bc
bx*c, bcx*, x*bc
```

但, 若我们在这些RE前后分别加 `^` 与 `$` 这样的anchor，那又如何呢？

刚学RE时，只要能掌握上面这些基本的meta的大概就可以入门了。 一如前述，RE是一种规范化的文字表达式， 主要用于某些文字处理工具之间，如： grep， perl， vi，awk，sed，等等， 常用于表示一段连续的字符串，查找和替换。

然而每种工具对RE表达式的具体解读或有一些细微差别， 不过节本原理还是一致的。 只要掌握RE的基本原理，那就一理通百理了， 只是在实践时，稍加变通即可。

比方以grep来说， 在Linux上，你可以找到grep，egrep，fgrep这些程序， 其差异大致如下：

grep： 传统的grep程序，在没有任何选项(options)的情况下，只输出符合RE字串的句子， 其常见的选项如下：

选项 **(option)**	用途
-v	反模式， 只输出“不含”RE的字符串的行。
-r	递归模式，可同时处理所有层级的子目录里的文件
-q	静默模式，不输出任何结果(stderr 除外，常用于获取return value，符合为true，否则，为false.
-i	忽略大小写
-w	整词匹配，类似 \
-n	同时输出行号
-l	输出匹配RE的文件名
-o	只输出匹配RE的字符串。(gnu新版独有，不见得所有版本支持)
-E	切换为egrep

egrep：为grep的扩充版本，改良了许多传统grep不能或者不便的操作，

- grep下不支持 `?` 与 `+` 这两种meta，但egrep支持；
- grep 不支持 `a|b` 或（ `abc|xyz` ）这类“或一”的匹配，但egrep支持；
- grep 在处理 `{n,m}` 时，需要\{ 与 \}处理，但egrep不需。

等诸如此类的。我个人建议能用egrep就不用grep啦...^_^

fgrep: 不作RE处理，表达式仅作一般的字符串处理，所有的meta均市区功能。

好了，关于RE的入门，我们暂时就介绍到这里。 虽然有点乱，且有些观念也不恨精确， 不过，姑且算是对大家的一个交差吧...^_^ 若这两天有时间的话，我在举些范例来分析一下，以帮助大家更好的理解。 假如更有可能的话，也顺道为大家介绍一下sed这个工具。

Part-III eval

讲到command line的重组特性， 真的需要我们好好的加以解释的。

如此便能抽丝剥茧的一层层的将整个command line分析的 一清二楚，而不至于含糊。

假如这个重组的特性理解了，那我们介绍一个好玩的命令：`eval`.

我们在变量替换的过程中，常会碰到所谓的复式变量的问题： 如：

```
a=1
A1=abc
```

我们都知道`echo $A1`就可以得到abc的结果。 然而，我们能否用Aa来取代$A1，而同一样替换为abc呢？

这个问题我们可用很轻松的用`eval`来解决：

```
eval echo \$A$a
```

说穿了，`eval` 只不过是在命令行完成替换重组后， 在来一次替换重组罢了... 就是这么简单啦～～～ ^_^

shell十三问之16：学习总结与原帖目录

本人(markdown译者)是解决工作中shell脚本的一个问题， 偶尔的一次机会遇到了CU论坛中这样一个神贴：**shell**十三问.

shell十三问是CU的shell版的台湾的网中人是2003年用繁体发布的。 第一次读到shell十三问，由于是繁体，第一感觉有点抵触， 但是还是耐着性子读完了一贴，没想到竟然读懂了，而且还被网中人的幽默的写作风格，独到的思维方式， 循序渐进的认识事物的过程所折服。

尽管帖子是10多年前写的，今天看来也几乎没有一点过时的感觉。 从这个方面来说，shell十三问应该shell的(思想)精华本质所在， 就像武功的内功心法，可能我说的点过， 但是我曾经看过一本shell脚本学习指南，看完后的感觉，还是有感念很朦胧， 而shell十三问是我最容易理解和接受的，这也是我整理的Markdown版本初衷。 为什么不让好东西让更多的人熟知呢，恰好年前项目管理开始迁移到git上， 在git上认识一个好东西Markdown，用它可以很简单地整理出条例清晰篇章。 在年假的时候，觉得这个假期该做点什么， 毕竟马总都说了，改变世界，不如改变自己。

本人整理的 [简体中文Markdown版本的shell十三问][shell-markdown] 的链接地址：
https://github.com/wzb56/13_questions_of_shell

网中人的CU原帖shell十三问地址：http://bbs.chinaunix.net/thread-218853-1-1.html

我简单将原文整理如下：

我在CU的日子并不长，有幸在shell版上与大家结缘。 除了跟前辈学习到不少技巧之外，也常看到不少朋友的问题。 然而，在众多问题中，我发现许多瓶颈都源于shell的基础而已。 每次要解说，却总有千言万语不知从何而起之感......

这次，我不是来回答，而是准备了关于shell基础的十三个问题要问大家。 希望的shell的学习者们能够通过寻找答案的过程，好好的将shell基础打扎实一点。

当然了，这些问题我也会逐一解说一遍。 只是，我不敢保证什么时候能够完成这趟任务。

除了时间关系外，个人功力实在有限，很怕匆忙间误导观众就糟糕了。 若能抛砖引玉，诱得，其他前辈出马补充，那才是功德一件。

shell十三问：

4. " "(双引号) 与 ' '(单引号)差在哪？ (2008-10-30 02:07 最后更新)
5. var=value 在export前后差在哪？ (2008-10-30 02:12 最后更新)
6. exec 跟 source 差在哪? (2008-10-30 02:17 最后更新)
7. ( ) 与 { } 差在哪？
8. $(( )) 与 $( ) 还有${ } 差在哪？ (2008-10-30 02:20 最后更新)
9. $@ 与 $* 差在哪？
10. && 与 || 差在哪？ (2008-10-30 02:21 最后更新)
11. > 与 < 差在哪？ (2008-10-30 02:24 最后更新)
12. 你要 if 还是 case 呢？ (2008-10-30 02:25最后更新)
13. for what? while 与 until 差在哪？ (2008-10-30 02:26最后更新)
14. 跟 [!] 差在哪？
15. Part-I: Wildcard (2008-10-30 02:25 最後更新)
16. Part-II Regular Expression (2008-10-30 02:26 最后更新)

说明：

1. 欢迎大家补充/扩充问题。
2. 我接触电脑的中文名称时是在台湾，因此一些术语或与大陆不同，请自行转换。
3. 我会不定时"逐题"说明(以 Linux 上的 bash 为环境) 同时，也会在任何时候进行无预警的修改。请读者自行留意。
4. 本人于本系列所发表的任文章均可自由以电子格式(非印刷)引用、修改、转载， 且不必注明出处(若能注明 CU 更佳)。当然，若有错漏或不当结果，本人也不负任何责任。
5. 若有人愿意整理成册且付印者，本人仅保留著作权，版权收益之 30% 須捐赠于 CU 论坛管理者，剩余不究。

建议参考谈论:

1. shaoping0330 兄关于变量替换的补充：（链接在改版后已经失效）
2. shaoping0330 兄关于 RE 的说明:

3. 关于 nested subshell 的讨论:（链接在改版后已经失效）
4. 关于 IFS 的讨论:

- 感谢 lkydeer 兄整理 word/pdf 版本方便大家参考：